A program logic for resources

Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 389 (2007) 411–445
Theoretical Computer Science
www.elsevier.com/locate/tcs

A program logic for resources

David Aspinall[a,*], Lennart Beringer[a], Martin Hofmann[b], Hans-Wolfgang Loidl[b], Alberto Momigliano[a]

[a] *Laboratory for the Foundations of Computer Science, School of Informatics, University of Edinburgh, Edinburgh EH9 3JZ, Scotland, United Kingdom*
[b] *Institut für Informatik, Ludwig-Maximilians Universität, D-80538 München, Germany*

Abstract

We introduce a reasoning infrastructure for proving statements about resource consumption in a fragment of the Java Virtual Machine Language (JVML). The infrastructure is based on a small hierarchy of program logics, with increasing levels of abstraction: at the top there is a type system for a high-level language that encodes resource consumption. The infrastructure is designed to be used in a proof-carrying code (PCC) scenario, where mobile programs can be equipped with formal evidence that they have predictable resource behaviour.

This article focuses on the core logic in our infrastructure, a VDM-style program logic for partial correctness, which can make statements about resource consumption alongside functional behaviour. We establish some important results for this logic, including soundness and completeness with respect to a resource-aware operational semantics for the JVML. We also present a second logic built on top of the core logic, which is used to express termination; it too is shown to be sound and complete. We then outline how high-level language type systems may be connected to these logics.

The entire infrastructure has been formalized in Isabelle/HOL, both to enhance the confidence in our meta-theoretical results, and to provide a prototype implementation for PCC. We give examples to show the usefulness of this approach, including proofs of resource bounds on code resulting from compiling high-level functional programs.

Keywords: Program logic; Proof-carrying-code; Object-oriented languages; Java virtual machine language; Cost modelling; Quantitative type-systems; Lightweight verification

1. Introduction

When we receive a program from another party, we want to know that it serves its intended purpose. Apart from having correct functional behaviour, it is important that a program meets non-functional requirements such as reasonable resource consumption. Indeed, non-functional requirements of application programs can often be *more* important than functional ones, as they impact on the overall security and robustness of a system. For example, a game

* Corresponding author.
E-mail addresses: da@inf.ed.ac.uk (D. Aspinall), lenb@inf.ed.ac.uk (L. Beringer), mhofmann@informatik.uni-muenchen.de (M. Hofmann), hwloidl@informatik.uni-muenchen.de (H.-W. Loidl), amomigl1@inf.ed.ac.uk (A. Momigliano).

0304-3975/$ - see front matter © 2007 Published by Elsevier B.V.
doi:10.1016/j.tcs.2007.09.003

to be run on a mobile telephone should not make excessive network demands or use up all the phone's memory; this is more important to the user and network operator than, say, that the game calculates the correct score.

We want to equip mobile code with guarantees that resource constraints are satisfied, following the proof-carrying code paradigm (PCC, [1]). PCC has emerged as a powerful means to guarantee type-correct behaviour or the adherence to various security policies. Low-level code is sent together with a certificate which contains a formal logical proof certifying that the program adheres to a given resource policy.

The bedrock of PCC is the logic for low-level code used to state and prove properties of interest. The logic must satisfy several crucial requirements. First, it must be expressive enough to state and prove the policies required for programs of concern. Second, it must be sound: proofs in the logic must be faithful to the behaviour of the code on the target machine. Third, we must meet engineering requirements for implementing proof-carrying code: we must have some way of effectively generating proofs in the logic, transmitting them in proof certificates, and efficiently checking them on receipt.

In this paper we present two program logics designed to meet these requirements, as part of a prototype proof-carrying code infrastructure aimed at certifying resource usage of Java bytecode programs.

Our core program logic is expressive enough to describe arbitrary cost metrics which are calculated by a resource-annotated semantics for a fragment of the Java Virtual Machine Language. We formalize this semantics as a big-step operational semantics whose correctness is self-evident. The core program logic makes partial correctness statements. For the annotated operational semantics, we prove that this logic is sound and (relative to the underlying assertion language) complete. Using the logic instead of the operational semantics directly has the main advantage of providing direct support for reasoning with invariants for recursive methods and loops (represented as tail recursive functions). To prove that a method satisfies some specification it suffices to show that its body satisfies it assuming that any recursive call does so. Proving this principle sound for the operational semantics requires some effort which would otherwise have to be spent with each individual verification.

We define a second logic for termination built on top of the core logic for partial correctness. Separating the two logics allows us to establish resource properties under the assumption of termination following one proof approach, and establish termination using a possibly different approach, and only where it is necessary (for example, on a case-by-case basis for individual methods). We also prove that the termination logic is sound and complete.

The main part of the paper is concerned with the definition and metatheoretical properties of the core logic and termination logic. Towards the end of the paper we will return to the original motivation of proof-carrying code, and describe how the logics can be fitted into a larger infrastructure based on type systems for high-level languages. The approach addresses the necessary engineering requirements for PCC.

This work represents a significant contribution to research in generalizing proof-carrying code scenario to richer policies and richer languages; to our knowledge it represents one of the first applications of PCC for resources to Java bytecode. It also provides advances in program logics for low-level code; we believe that our method has a number of advantages over many existing related approaches, including the good meta-theoretical properties of the logics, separate treatment of (partial) correctness and termination, and the functional treatment of iteration and local variables. The body of the paper and a closing comparison with the existing literature describes these and other advantages in detail. The entire infrastructure has been formalized in the theorem prover Isabelle/HOL, both to enhance confidence in the meta-theoretical results, and to provide a prototype implementation for PCC. This contribution is part of our work on the *Mobile Resource Guarantees* (MRG) project [2], see [3] for a high-level overview of the whole project.

1.1. Outline

The structure of this paper is as follows. Section 2 presents the *Grail* language for which we build these logics. Grail is a subset of the Java Virtual Machine Language (JVML), written using a functional notation. The operational semantics of Grail is annotated with a mechanism for calculating resource costs, using operators for each expression former.

Section 3 presents the core program logic for partial correctness for Grail. It allows an expressive form of assertion that facilitates compositional reasoning and gives powerful rules for mutual recursion. Proofs of soundness and completeness for this logic are given, based on a full formalization in Isabelle/HOL. Examples demonstrate the use of this logic for proving resource properties, making use of special rules for mutually recursive program fragments and method invocations with argument adaptation.

Section 4 builds the termination logic on top of the partial correctness logic, and thus arrives at a total correctness logic. Proofs of soundness and completeness are given. An earlier example is revisited to demonstrate the methodology.

In Section 5 we return to the big picture and describe how our hierarchy of program logics is used in the *Mobile Resource Guarantees* architecture to obtain a system with a certifying compiler and independent validation of the generated certificate. Section 6 discusses related work on program logics and their formalizations. Finally, Section 7 summarizes the main results of our work and sketches directions for further work.

2. The Grail language

Rather than working with the JVML directly, we use a reformulation called *Grail* (a contraction of *Guaranteed Resource Aware Intermediate Language*). The version of Grail studied in this paper covers the most important features of the JVML, including boolean and integer values, reference values and objects, mutable fields (allowing aliasing), static and instance methods. We do not cover exceptions, arrays or threads here.

The design of Grail was motivated by the following goals, which we elaborate in turn:

(1) Suitability as a target for a resource-transparent compiler from a functional high-level language;
(2) Proximity to standard mobile code formats, so as to obtain executability and code mobility using existing wire formats;
(3) Suitability as a basis for attaching resource assertions;
(4) Amenability to formal proofs about resource usage.

Grail uses a functional notation that abstracts from some JVM specific details such as the use of numerical variable names as locals, and it has the operational semantics of an impure functional language: mutable object fields, immutable local variables, and tail recursive functions instead of iteration. This makes it ideally suitable for compilation of high-level functional languages such as *Camelot*, which is a variant of OCaml with a resource-aware type system [4].

Code in Grail remains close to standard formats; it can be reversibly expanded into (a subset of) virtual machine languages including JVM, the JCVML of JavaCard or the MSIL used by .NET.[1]

Because the translation mechanism into a virtual machine language is fixed [5], the resource usage of Grail programs can be captured by considering the cost of execution of the translated form, and those costs can be used to instantiate a resource model which is part of the operational semantics.

Our program logic and its implementation described in Sections 3 and 4 provide some evidence that the final design goal of Grail has indeed been achieved, namely that it is amenable to formal proofs.

2.1. Syntax

At the level of classes and methods, the Grail representation of code retains the syntactic structure of Java bytecode. Method bodies, in contrast, are represented as collections of mutually tail-recursive A-normalized functions (ANF) [6]. In particular, only primitive operations can occur as e_1 in a let-expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$. Similar to the λ-JVM [7] and other functional intermediate languages [8], primitive instructions model arithmetic operations, object manipulation (object creation, field access) and the invocation of methods. Strengthening the syntactic conditions of ANF, actual arguments in function calls are required to coincide syntactically with the formal parameters of the function definitions. This condition allows function calls to be interpreted as immediate jump instructions; register shuffling at basic block boundaries is performed by the calling code rather than being built into the function application rule. As a result, the consumption of resources at virtual machine level may be expressed in a functional semantics for Grail: the expansion into bytecode does not require register allocation or the insertion of gluing code [5]. In contrast to the aim of λ-JVM, Grail does not aim to represent arbitrary bytecode. Instead, the translation of raw bytecode into Grail is only defined on the image of our expansion — and on this subset, it is the reversal of the code expansion. As each Grail instruction form expands to a sequence of bytecode instructions that leaves the operand stack empty, the

[1] At present, our tools cover only the JVML.

compiled code is highly structured. In particular, it satisfies all conditions identified by Leroy [9] as being required for efficient bytecode verification on smart cards and other resource-constrained devices.

For the purpose of this paper, the syntactic restrictions of Grail may be largely ignored, since their significance concerns the relationship between Grail and actual bytecode rather than the program logic, whose correctness does not require them. The formal syntax treated in the remainder of this article therefore comprises a single category of expressions *expr* that is defined over mutually disjoint sets $\mathcal{M}$ of method names, $\mathcal{C}$ of class names, $\mathcal{F}$ of function names (i.e. labels of basic blocks), $\mathcal{T}$ of (static) field names and $\mathcal{X}$ of variables. These categories are, respectively, ranged over by m, c, f, t, and x. In addition, i ranges over immediate values (integers, and booleans true and false) and *op* over primitive operation of type $\mathcal{V} \Rightarrow \mathcal{V} \Rightarrow \mathcal{V}$ such as arithmetic operations or comparison operators, where $\mathcal{V}$ is the semantic category of values. Values are ranged over by v and comprise immediate values, references r, and the special symbol $\perp$, which stands for the absence of a value. References are either null or of the form Ref l where $l \in \mathcal{L}$ is a location.

The syntax is given by the grammar below, which defines expressions *expr* and method arguments *args*. We write $\overline{a}$ to stand for lists of arguments.

$$
\begin{aligned}
e \in \mathit{expr} \;::=\; & \texttt{null} \mid \texttt{imm}\; i \mid \texttt{var}\; x \mid \texttt{prim}\; \mathit{op}\; x\; x \mid \texttt{new}\; c\; [t_1 := x_1, \ldots, t_n := x_n] \mid \\
& x.t \mid x.t{:=}x \mid c \diamond t \mid c \diamond t{:=}x \mid \texttt{let}\; x = e\; \texttt{in}\; e \mid e\; ;\; e \mid \\
& \texttt{if}\; x\; \texttt{then}\; e\; \texttt{else}\; e \mid \texttt{call}\; f \mid x \cdot m(\overline{a}) \mid c.m(\overline{a}) \\
a \in \mathit{args} \;::=\; & \texttt{var}\; x \mid \mathsf{null} \mid i.
\end{aligned}
$$

Our notations here are intentionally somewhat prolix, for example, by including the injections Ref, `var` and `imm`, which would ordinarily be omitted using meta-syntactic conventions for names. We hope that the reader will forgive the additional verbosity in return for the reassurance that the following rules, programs and example verifications have all been formalized precisely as they are written here.

Expressions $e \in \mathit{expr}$ represent basic blocks and are built from operators, constants and previously computed values. Expressions correspond to primitive sequences of bytecode instructions that may, as a side-effect, alter the heap. For example, $x.t$ and $x.t{:=}y$ represent (non-static) `getfield` and `putfield` instructions, while $c \diamond t$ and $c \diamond t{:=}y$ denote their static counterparts. The binding $\texttt{let}\; x = e_1\; \texttt{in}\; e_2$ is used if the evaluation of e_1 returns a value on top of the JVM stack while $e_1\; ;\; e_2$ represents purely sequential composition, used, for example, if e_1 is a field update $x.t{:=}y$. Object creation includes the initialization of the object fields according to the argument list: the content of variable x_i is stored in field t_i. Function calls (`call`) follow the Grail calling convention (i.e. correspond to immediate jumps) and do not carry arguments. The instructions $x \cdot m(\overline{a})$ and $c.m(\overline{a})$ represent virtual (instance) and static method invocation, respectively. The keyword `self` is a reserved variable name standing for the current object (`this` in Java).

A formal type system can be imposed on Grail programs to rule out bad programs by reflecting typing conditions enforced by the underlying virtual machine. Our operational semantics and program logic are more general, although we always consider well-typed programs.

In the theoretical development, we assume that all method declarations employ distinct names for identifying inner basic blocks. A program P consists of a table FT mapping each function identifiers to an expression, and a table MT associating a list of method parameters (i.e. variables) and an expression (the initial basic block) to class names and method identifiers. We use the notations body_f and $\mathit{body}_{c,m}$ to denote the bodies of function f and method $c.m$, respectively, and $\mathit{pars}_{c,m}$ to denote the formal parameters of $c.m$.

Fig. 1 shows an example method for appending the list represented by argument l_2 to the list l_1. The Grail code shown is the pretty-printed output of a compiler run on the source code: fragment

```
type ilist = Nil | Cons of int * ilist
let append l1 l2 = match l1 with
                     Nil => l2
                   | Cons(h,t) => Cons(h,append t l2)
```

which is written in the high-level functional language Camelot [10,11]. Integer lists are represented by objects of class LIST with fields HD, TL, and TAG, where TAG is used to discriminate between the constructors, i.e. empty

$$
\begin{array}{l}
\texttt{method}\ \mathsf{LIST}\ \mathsf{LIST.append}(\mathsf{l}_1, \mathsf{l}_2) = \texttt{call}\ f \\
\left[\begin{array}{lcl}
f & \mapsto & \texttt{let}\ \mathsf{tg} = \mathsf{l}_1.\mathsf{TAG}\ \texttt{in} \\
 & & \texttt{let}\ \mathsf{b} = \texttt{prim}\ (\lambda\ z\ y.\ \text{if}\ z = 1\ \text{then}\ \mathsf{true}\ \text{else}\ \mathsf{false})\ \mathsf{tg}\ \mathsf{tg}\ \texttt{in} \\
 & & \texttt{if}\ \mathsf{b}\ \texttt{then}\ \texttt{var}\ \mathsf{l}_2\ \texttt{else}\ \texttt{call}\ f_1 \\
f_1 & \mapsto & \texttt{let}\ \mathsf{h} = \mathsf{l}_1.\mathsf{HD}\ \texttt{in}\ \texttt{let}\ \mathsf{t} = \mathsf{l}_1.\mathsf{TL}\ \texttt{in} \\
 & & \texttt{let}\ \mathsf{l}_1 = \mathsf{LIST.append}([\texttt{var}\ \mathsf{t}, \texttt{var}\ \mathsf{l}_2])\ \texttt{in}\ \texttt{let}\ \mathsf{tg} = \texttt{imm}\ 2\ \texttt{in} \\
 & & \texttt{new}\ \mathsf{LIST}\ [\mathsf{TAG} := \mathsf{tg}, \mathsf{HD} := \mathsf{h}; \mathsf{TL} := \mathsf{l}_1]
\end{array}\right]
\end{array}
$$

Fig. 1. Code of method append.

($\mathsf{TAG} = 1$) and non-empty ($\mathsf{TAG} = 2$) lists. The emitted code consists of a method containing two functions whose bodies are shown; the body of the method itself is the term $\texttt{call}\ f$ which appears on the first line.[2]

2.2. Resource algebras

To admit reasoning about allocation and consumption of different computational resources, our operational semantics is annotated with a resource counting mechanism based on a general cost model provided by a notion of *resource algebra*. A resource algebra provides a basis for quantitative measurements such as instruction counters, but it can also be used to monitor cost-relevant events such as the allocation of fresh memory, calls to specific methods, or the maximal height of the frame stack encountered during the execution of a program. Moreover, the resources are a purely non-invasive annotation on the ordinary operational semantics; evaluation of an expression is not affected by the resources consumed in subexpressions.

Definition 1. A resource algebra $\mathcal{R}$ has a carrier set R consisting of *resource values* $r \in R$, together with:

- A cost ordering $\leq \subseteq R \times R$
- For the atomic expressions, families of constants $\mathcal{R}^{\texttt{null}}$, $\mathcal{R}^{\texttt{imm}}_{i}$, $\mathcal{R}^{\texttt{var}}_{x}$, $\mathcal{R}^{\texttt{prim}}_{op,x,y}$, $\mathcal{R}^{\texttt{getf}}_{x,t}$, $\mathcal{R}^{\texttt{putf}}_{x,t,y}$, $\mathcal{R}^{\texttt{gets}}_{c,t}$, $\mathcal{R}^{\texttt{puts}}_{c,t,y}$, $\mathcal{R}^{\texttt{new}}_{c,x_1,\ldots,x_n}$ of type R,
- For compound expressions, families of operations $\mathcal{R}^{\texttt{if}}_{x}$, $\mathcal{R}^{\texttt{call}}_{f}$, $\mathcal{R}^{\texttt{invs}}_{c,m,\bar{a}}$ and $\mathcal{R}^{\texttt{invv}}_{x,m,\bar{a}}$ of type $R \to R$, and
- $\mathcal{R}^{\texttt{let}}_{x}$, $\mathcal{R}^{\texttt{comp}}$ of type $(R \times R) \to R$.

Resource algebras simply collect together constant costs and operations on costs for each expression former in the syntax. The cost ordering expresses when one resource value is considered cheaper or better than another. Our operational semantics and logics are then based on an arbitrary resource algebra.[3]

An alternative way of cost accounting used elsewhere is by *instrumenting* the original code, i.e. by inserting additional program variables and instructions to maintain resource counters, but without otherwise changing the underlying program. Then the costs of executing the original program are obtained by reasoning about (or executing, or executing symbolically, or performing static analysis on) the instrumented program [14–16]. Although flexible, instrumentation has certain disadvantages. Most obviously, if the instrumented code is executed at run-time, there will be extra costs associated in book-keeping and resource usage patterns themselves may change compared with the uninstrumented code. Executed or not, in general there is a risk that the program behaviour of instrumented code may differ from the original code due to race conditions. And compared with wholly static approaches, if run-time monitors are employed to enforce a resource policy, we risk wasting computational resources by terminating offending computations before they produce useful results. Finally and fundamentally, instrumentation limits the costs one may

[2] In later examples, we do simplify the notation somewhat, using some hopefully obvious shorthand from our implementation for standard primitive operations and omitting the prim tag, for example, $\texttt{let}\ \mathsf{b} = \textit{iszero}\ \mathsf{v}_3\ \texttt{in}\ \ldots$.

[3] We have investigated the algebraic structure of resources elsewhere [13]. A similar notion, under the name of *Kripke resource monoid*, has been independently used as the basis of the semantics of the logic of Bunched Implication [12].

	$\mathcal{R}^{\mathsf{Count}}$	$\mathcal{R}^{\mathsf{InvTr}}$	$\mathcal{R}^{\mathsf{PVal}}(\mathsf{C}, \mathsf{M}, P)$	$\mathcal{R}^{\mathsf{HpTr}}$
R	$\mathcal{N} \times \mathcal{N} \times \mathcal{N} \times \mathcal{N}$	$\overline{expr}$	$\mathcal{B}$	$\overline{\mathcal{H}}$
$\mathcal{R}^{\mathtt{null}}$	$\langle 1\,0\,0\,0\rangle$	$[\,]$	true	$[h]$
$\mathcal{R}^{\mathtt{imm}}_{i}$	$\langle 1\,0\,0\,0\rangle$	$[\,]$	true	$[h]$
$\mathcal{R}^{\mathtt{var}}_{x}$	$\langle 1\,0\,0\,0\rangle$	$[\,]$	true	$[h]$
$\mathcal{R}^{\mathtt{prim}}_{op,x,y}$	$\langle 3\,0\,0\,0\rangle$	$[\,]$	true	$[h]$
$\mathcal{R}^{\mathtt{new}}_{c,x_1,\ldots,x_n}$	$\langle (n+1)\,0\,0\,0\rangle$	$[\,]$	true	$[h]$
$\mathcal{R}^{\mathtt{getf}}_{x,t}$	$\langle 2\,0\,0\,0\rangle$	$[\,]$	true	$[h]$
$\mathcal{R}^{\mathtt{putf}}_{x,t,y}$	$\langle 3\,0\,0\,0\rangle$	$[\,]$	true	$[h]$
$\mathcal{R}^{\mathtt{gets}}_{c,t}$	$\langle 2\,0\,0\,0\rangle$	$[\,]$	true	$[h]$
$\mathcal{R}^{\mathtt{puts}}_{c,t,y}$	$\langle 3\,0\,0\,0\rangle$	$[\,]$	true	$[h]$
$\mathcal{R}^{\mathtt{let}}_{x}(p,q)$	$\langle 1\,0\,0\,0\rangle \oplus (p \sqcap q)$	$p@q$	$p \wedge q$	$p@q$
$\mathcal{R}^{\mathtt{comp}}(p,q)$	$p \sqcap q$	$p@q$	$p \wedge q$	$p@q$
$\mathcal{R}^{\mathtt{if}}_{x}(p)$	$\langle 2\,0\,0\,0\rangle \oplus p$	p	true	p
$\mathcal{R}^{\mathtt{call}}_{f}(p)$	$\langle 1\,1\,0\,0\rangle \oplus p$	p	true	p
$\mathcal{R}^{\mathtt{invv}}_{x,m,\overline{a}}(p)$	$\langle (4+\lvert\overline{a}\rvert)\,0\,1\,1\rangle \oplus p$	$(x \cdot m(\overline{a})) :: p$	p	p
$\mathcal{R}^{\mathtt{invs}}_{c,m,\overline{a}}(p)$	$\langle (2+\lvert\overline{a}\rvert)\,0\,1\,1\rangle \oplus p$	$(c.m(\overline{a})) :: p$	$((c = \mathsf{C} \wedge m = \mathsf{M}) \longrightarrow P(\overline{a})) \wedge p$	p

Fig. 2. Four example resource algebras.

consider to quantities where the domain and the modifying operations are expressible in the programming language, and the interaction between resource-counting and "proper" variables may become difficult to reason about. Our approach avoids these problems by including structured costs in the operational semantics. It retains flexibility by being parametrized on the form of the costs, and, if required, semantics costs could be formally related to suitably instrumented code using our program logic.

2.3. Resource algebra examples

A few motivating examples of resource algebras are collected in Fig. 2. The first algebra, $\mathcal{R}^{\mathsf{Count}}$, represents a simple cost model of the JVM with four metrics, and was already presented in [17]. The first component of a tuple models an instruction counter that approximates execution time. Charging all JVM instructions at the same rate, this counter is incremented roughly by the number of bytecode instructions each expression is expanded to. For example, a field modification $x.t{:=}y$ expands to a bytecode sequence of length three, comprising two load operations that copy the contents of variables x and y onto the operand stack, and one `putfield` operation. In the rules for method invocations, we charge for pushing all arguments onto the operand stack, the invocation and the returning instruction, plus, in the case of a virtual invoke, for loading the object reference and the virtual method lookup. The second and third components represent more specific instruction counters, for function calls (jumps) and method invocations, respectively. Naturally, more fine-grained instruction counters can easily be defined, for example, by counting the invocation of different methods separately or by charging complex instructions at an increased rate. Finally, the fourth component monitors the maximal frame stack height observed during a computation. This value is non-zero exactly for the frame-allocating instructions $x \cdot m(\overline{a})$ and $c.m(\overline{a})$.

The definition of $\mathcal{R}^{\mathsf{Count}}$ uses two binary operations, $\oplus$ and $\sqcap$, to combine costs. The former represents pointwise addition in all four components, while the latter performs pointwise addition in the first three components, and the *max* operation in the fourth component. Note that we could have formulated the counters as four separate resource algebras and obtained $\mathcal{R}^{\mathsf{Count}}$ as their product.

Some of our main examples are concerned with the amount of heap space used during execution of a program. To monitor the dynamic allocation of objects we could use a counter in $\mathcal{R}^{\mathsf{Count}}$ for the instruction `new`, but since the

operational semantics does not model garbage collection, we can in fact infer the number and class of objects allocated by comparing the initial and final heaps.

The second resource algebra, $\mathcal{R}^{\mathsf{InvTr}}$, ranges over expression sequences and collects the (static or virtual) method invocations in execution order. Here, the value for each nullary expression former is the empty list; branches and function invocations take the value for subexecutions; invocations prefix the respective method calls to the sequence (notation :: in the meta-logic), and the operations for binary program composition $\mathcal{R}^{\mathtt{let}}_{x}$ and $\mathcal{R}^{\mathtt{comp}}$ append (notation @) the sequences in the order of evaluation of the subexpressions.

The last two resource algebras, $\mathcal{R}^{\mathsf{PVal}}(\mathsf{C}, \mathsf{M}, P)$ and $\mathcal{R}^{\mathsf{HpTr}}$, concern a slight generalization of the formal setup, where, in addition to pieces of syntax, values of resource algebras may also depend upon other components of the operational semantics (to be defined shortly), such as environments E and heaps h. This allows us to formulate policies that depend on the dynamic evaluation of syntactic items. The *parameter values* algebra $\mathcal{R}^{\mathsf{PVal}}(\mathsf{C}, \mathsf{M}, P)$ can be parametrized by a class name C, a method name M, and a *parameter policy* P which may refer to the environment and heap. The resource value in this algebra maintains a flag which acts as a monitor to ensure that each invocation of $\mathsf{C}.\mathsf{M}(\overline{a})$ satisfies the policy $P(\overline{a})$. For example, for each $k \in \mathcal{N}$, the policy

$$P_k(\overline{a}) \equiv \exists\, x\, n\, X.\, \overline{a} = [\mathtt{var}\ x] \ \wedge\ h \models_{list(n,X)} E\langle x\rangle \ \wedge\ n \leq k$$

stipulates that $\overline{a}$ consists of a single variable that represents a list of length at most k.[4] Value-constraining policies like this may be useful in the domain of embedded systems, where calls to external actuators must obey strict parameter limitations. See [3] for more motivation and a detailed example.

In general, resource algebras that depend on semantic components are sufficiently powerful to collect diagnostic traces along the chosen path of computation — the final example, $\mathcal{R}^{\mathsf{HpTr}}$ simply collects all intermediate heaps (the value set $\mathcal{H}$ is defined below). Like a syntactic trace originating from the algebra $\mathcal{R}^{\mathsf{InvTr}}$, the resulting word may be constrained by further policies, specified, for example, by security automata [18] or formulae from logics over linear structures which can be encoded in our higher-order assertion language.

When not considering specific examples, we understand $\mathcal{R}$ to represent a fixed, but arbitrary resource algebra, and identify $\mathcal{R}$ with its carrier set.

2.4. Operational semantics

The formal basis of the program logic is a big-step operational semantics that models an (impure) functional interpretation of Grail. Judgements relate expressions e to environments $E \in \mathcal{E}$, initial and final heaps $h, h' \in \mathcal{H}$, result values $v \in \mathcal{V}$ and costs $p \in \mathcal{R}$. Environments are maps from variables to values; we write $E\langle x\rangle$ for the value bound to x in E. Heaps consist of two components, an object heap and a class heap. The object heap is a finite map of type $\mathcal{L} \rightarrow_{fin} \mathcal{C} \times (\mathcal{T} \rightarrow_{fin} \mathcal{V})$, i.e. an object consists of a class name and a field table. The class heap stores the content of static fields and is represented by a map $\mathcal{C} \rightarrow_{fin} \mathcal{T} \rightarrow_{fin} \mathcal{V}$. The following operations are used for heaps:

$h(l).t$	field lookup of value at t in object at l,
$h[l.t \mapsto v]$	field update of t with v at l,
$h(c).t$	lookup for static field t in class c,
$h[c.t \mapsto v]$	update of static field t in class c,
$h(l)$	class name of object at l,
$dom\ h$	domain of the heap h,
$freshloc(h)$	a location l chosen so that $l \notin dom\ h$.

A judgement $E \vdash h, e \Downarrow h', v, p$ reads "in variable environment E and initial heap h, code e evaluates to the value v, yielding the heap h' and consuming p resources". The rules defining our semantics are formulated relative

[4] The definition of the predicate $h \models_{list(n,X)} E\langle x\rangle$ is given later, in Section 3.5.2.

to a fixed program P, whose components are accessed in the rules for functional calls and method invocations. The following rules use further notation which is explained in the commentary below.

$$\frac{}{E \vdash h, \mathtt{null} \Downarrow h, \mathsf{null}, \mathcal{R}^{\mathtt{null}}} \text{ (NULL)} \qquad \frac{}{E \vdash h, \mathtt{imm}\ i \Downarrow h, i, \mathcal{R}^{\mathtt{imm}}_{i}} \text{ (IMM)}$$

$$\frac{}{E \vdash h, \mathtt{var}\ x \Downarrow h, E\langle x\rangle, \mathcal{R}^{\mathtt{var}}_{x}} \text{ (VAR)}$$

$$\frac{}{E \vdash h, \mathtt{prim}\ op\ x\ y \Downarrow h, op\ (E\langle x\rangle)\ (E\langle y\rangle), \mathcal{R}^{\mathtt{prim}}_{op,x,y}} \text{ (PRIM)}$$

$$\frac{E\langle x\rangle = \mathsf{Ref}\, l}{E \vdash h, x.t \Downarrow h, h(l).t, \mathcal{R}^{\mathtt{getf}}_{x,t}} \text{ (GETF)} \qquad \frac{E\langle x\rangle = \mathsf{Ref}\, l}{E \vdash h, x.t{:=}y \Downarrow h[l.t \mapsto E\langle y\rangle], \bot, \mathcal{R}^{\mathtt{putf}}_{x,t,y}} \text{ (PUTF)}$$

$$\frac{}{E \vdash h, c \diamond t \Downarrow h, h(c).t, \mathcal{R}^{\mathtt{gets}}_{c,t}} \text{ (GETS)} \qquad \frac{}{E \vdash h, c \diamond t{:=}y \Downarrow h[c.t \mapsto E\langle y\rangle], \bot, \mathcal{R}^{\mathtt{puts}}_{c,t,y}} \text{ (PUTS)}$$

$$\frac{l = \mathit{freshloc}(h)}{E \vdash h, \mathtt{new}\ c\ [\overline{t_i := x_i}] \Downarrow h[l \mapsto (c, \{\overline{t_i := E\langle x_i\rangle}\})], \mathsf{Ref}\, l, \mathcal{R}^{\mathtt{new}}_{c,x_1,\ldots,x_n}} \text{ (NEW)}$$

$$\frac{E\langle x\rangle = \mathsf{true} \quad E \vdash h, e_1 \Downarrow h_1, v, p}{E \vdash h, \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Downarrow h_1, v, \mathcal{R}^{\mathtt{if}}_{x}(p)} \text{ (IFTRUE)}$$

$$\frac{E\langle x\rangle = \mathsf{false} \quad E \vdash h, e_2 \Downarrow h_1, v, p}{E \vdash h, \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Downarrow h_1, v, \mathcal{R}^{\mathtt{if}}_{x}(p)} \text{ (IFFALSE)}$$

$$\frac{E \vdash h, e_1 \Downarrow h_1, w, p \quad w \neq \bot \quad E\langle x := w\rangle \vdash h_1, e_2 \Downarrow h_2, v, q}{E \vdash h, \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \Downarrow h_2, v, \mathcal{R}^{\mathtt{let}}_{x}(p, q)} \text{ (LET)}$$

$$\frac{E \vdash h, e_1 \Downarrow h_1, \bot, p \quad E \vdash h_1, e_2 \Downarrow h_2, v, q}{E \vdash h, e_1\ ;\ e_2 \Downarrow h_2, v, \mathcal{R}^{\mathtt{comp}}(p, q)} \text{ (COMP)}$$

$$\frac{E \vdash h, \mathit{body}_f \Downarrow h_1, v, p}{E \vdash h, \mathtt{call}\ f \Downarrow h_1, v, \mathcal{R}^{\mathtt{call}}_{f}(p)} \text{ (CALL)}$$

$$\frac{\mathit{Env}(\mathtt{self} :: \mathit{pars}_{c,m}, \mathsf{null} :: \overline{a}, E) \vdash h, \mathit{body}_{c,m} \Downarrow h_1, v, p}{E \vdash h, c.m(\overline{a}) \Downarrow h_1, v, \mathcal{R}^{\mathtt{invs}}_{c,m,\overline{a}}(p)} \text{ (SINV)}$$

$$\frac{E\langle x\rangle = \mathsf{Ref}\, l \quad h(l) = c \quad \mathit{Env}(\mathtt{self} :: \mathit{pars}_{c,m}, x :: \overline{a}, E) \vdash h, \mathit{body}_{c,m} \Downarrow h_1, v, p}{E \vdash h, x \cdot m(\overline{a}) \Downarrow h_1, v, \mathcal{R}^{\mathtt{invv}}_{x,m,\overline{a}}(p)} \text{ (VINV)}$$

The first nine rules are rather straightforward — the costs are simply obtained by applying the corresponding components of the resource algebra to the syntactic components of the operation in question. In rules PUTF and PUTS, the return values are set to $\bot$, in accordance with the fact that the corresponding virtual machine codes do not leave a value on the operand stack. In the rule NEW, the allocated object is initialized by assigning the content of variable x_i to field t_i, for all fields.

The two rules for conditionals contain no surprises either — the resources consumed during the execution of the respective branch are promoted to the conclusion, adjusted by the operation $\mathcal{R}^{\mathtt{if}}_{x}$. Similarly, the rules for program composition, LET and COMP, combine the costs of the constituent expressions. The notation $E\langle x := w\rangle$ denotes

environment update. The reason why we do *not* define COMP in terms of LET lies in the different resource consumption of the two constructors.

In the rules CALL, SINV and VINV, the function and method bodies, and the method parameters, are retrieved from the (implicit) program P. As discussed earlier, function calls occur in tail position and correspond to jump instructions, hence the body is executed in the same environment. The costs of the function call are taken into account at the end of the execution. In contrast, method calls can occur in non-tail positions, hence the rules for static and virtual method invocations execute the method body in an unaltered heap, but in an environment that represents a new frame. The semantic function $Env(\overline{x}, \overline{a}, E)$ constructs a fresh environment that maps parameter x_i to the result of evaluating a_i in environment E, i.e. to $E\langle x\rangle$ if a_i is a variable x and to the explicit reference or integer value otherwise. The content of the `self` variable is set to the location of the invoking object in the case of rule VINV, and to the null value in rule SINV.

3. Program logic for partial correctness

The basis for reasoning and certificate generation is a general-purpose program logic for Grail where assertions are boolean functions over all semantic components occurring in the operational semantics, i.e. evaluation environments, pre- and post-heaps, result values, and values from a resource algebra. In this section, we define a logic of partial correctness (i.e. in particular, non-terminating programs satisfy any assertion), which is complemented by a termination logic in Section 4.

3.1. Assertions and validity

Deviating from the syntactic separation into pre- and post-conditions typical for Hoare-style and VDM-style program logics [19,20], a judgement in our logic relates a Grail expression e to a single assertion A

$$\Gamma \rhd e : A$$

dependent on a context

$$\Gamma = \{(e_1, A_1), \ldots, \{e_n, A_n)\}$$

which stores verification assumptions for recursive program structures.[5]

Following the so-called "shallow embedding" style, we encode assertions as predicates in the formal higher-order meta-logic (i.e. Isabelle/HOL). Assertions range over the components of the operational semantics, namely the input environment E and initial heap h, and the post heap h', the result value v, and the resources consumed p. An assertion A thus belongs to the type

$$\mathcal{A} \equiv \mathcal{E} \to \mathcal{H} \to \mathcal{H} \to \mathcal{V} \to \mathcal{R} \to \mathcal{B}$$

where $\mathcal{B}$ is the type of booleans. We use the notation of Isabelle/HOL for writing logical connectives and predicates, in particular, using λ-notation to define predicates:

$$A = \lambda\, E\ h\ h'\ v\ p.\ \cdots$$

and curried function application to denote application of predicates to particular semantic values:

$$A\ E_1\ h_1\ h'_1\ v_1\ p_1.$$

In the rules of the logic, the conclusions define assertions which hold for each form of expression, by applying assertions from the premises to appropriately modified values corresponding to the operational semantics. Axioms define assertions which are satisfied exactly by the corresponding evaluation in the semantics.

Although this assertion format may be unusual, it has advantages. Compared to program logics with pre- and post-conditions, a single assertion allows us to simplify the treatment of auxiliary variables: there is no need for special

[5] Later on we sometimes use the term "specification" as a synonym for "assertion", especially when referring to assumptions or assertions used to define behaviour exactly.

variables to record parts of the state before execution because the whole pre-state is available to the assertion at the same time as the post-state. The single assertion format also gives a more elegant rule for composition, without the modified precondition which is typical for Hoare-style logics. We discuss this further in Section 3.3.

The validity of assertion A for expression e is defined by a partial correctness interpretation: A must be satisfied for all terminating executions of e.

Definition 2 (*Validity*). Assertion A is *valid* for e, written $\models e : A$, if

$$E \vdash h, e \Downarrow h', v, p \quad \text{implies } A\ E\ h\ h'\ v\ p$$

for all E, h, h', v, and p.

This definition may be lifted to contexts Γ in the obvious way.

Definition 3 (*Contextual Validity*). Context Γ is *valid*, notation $\models \Gamma$, if all pairs (e, A) in Γ satisfy $\models e : A$. Assertion A is *valid for e in context* Γ, written $\Gamma \models e : A$, if $\models \Gamma$ implies $\models e : A$.

We next turn to the description of our proof system and the proof of its soundness and completeness for this notion of validity.

3.2. Proof system

The program logic comprises one rule for each expression form, and two logical rules, VAX and VCONSEQ. Again, we consider classes and methods for a fixed program P.

$$\frac{}{\Gamma \rhd \mathtt{null} : \lambda\, E\, h\, h'\, v\, p.\, h' = h \ \wedge\ v = \mathsf{null} \ \wedge\ p = \mathcal{R}^{\mathtt{null}}} \quad \text{(VNULL)}$$

$$\frac{}{\Gamma \rhd \mathtt{imm}\, i : \lambda\, E\, h\, h'\, v\, p.\, h' = h \ \wedge\ v = i \ \wedge\ p = \mathcal{R}^{\mathtt{imm}}_{i}} \quad \text{(VIMM)}$$

$$\frac{}{\Gamma \rhd \mathtt{var}\, x : \lambda\, E\, h\, h'\, v\, p.\, h' = h \ \wedge\ v = E\langle x\rangle \ \wedge\ p = \mathcal{R}^{\mathtt{var}}_{x}} \quad \text{(VVAR)}$$

$$\frac{}{\Gamma \rhd \mathtt{prim}\, op\, x\, y : \lambda\, E\, h\, h'\, v\, p.\ \ v = op\, E\langle x\rangle\ E\langle y\rangle\ \ \wedge\ h' = h \ \wedge\ p = \mathcal{R}^{\mathtt{prim}}_{op,x,y}} \quad \text{(VPRIM)}$$

$$\frac{}{\Gamma \rhd x.t : \lambda\, E\, h\, h'\, v\, p.\, \exists l.\ \ E\langle x\rangle = \mathsf{Ref}\, l \ \wedge\ h' = h\ \ \wedge\ v = h'(l).t\ \ \wedge\ p = \mathcal{R}^{\mathtt{getf}}_{x,t}} \quad \text{(VGETF)}$$

$$\frac{}{\begin{array}{l}\Gamma \rhd x.t{:=}y : \lambda\, E\, h\, h'\, v\, p.\, \exists l.\ \ E\langle x\rangle = \mathsf{Ref}\, l \ \wedge\ p = \mathcal{R}^{\mathtt{putf}}_{x,t,y} \wedge \\ \qquad\qquad h' = h[l.t \mapsto E\langle y\rangle]\ \ \wedge\ v = \bot\end{array}} \quad \text{(VPUTF)}$$

$$\frac{}{\Gamma \rhd c \diamond t : \lambda\, E\, h\, h'\, v\, p.\, h' = h\ \ \wedge\ v = h(c).t\ \ \wedge\ p = \mathcal{R}^{\mathtt{gets}}_{c,t}} \quad \text{(VGETST)}$$

$$\frac{}{\Gamma \rhd c \diamond t{:=}y : \lambda\, E\, h\, h'\, v\, p.\, h' = h[c.t \mapsto E\langle y\rangle]\ \ \wedge\ v = \bot \ \wedge\ p = \mathcal{R}^{\mathtt{puts}}_{c,t,y}} \quad \text{(VPUTST)}$$

$$\frac{}{\Gamma \rhd \texttt{new}\ c\ [\overline{t_i := x_i}] : \lambda\, E\ h\ h'\ v\ p.\ \exists l.\ \ l = \mathit{freshloc}(h)\ \wedge\ p = \mathcal{R}^{\texttt{new}}_{c,x_1,\ldots,x_n} \wedge h' = h[l \mapsto (c, \{\overline{t_i := E\langle x_i\rangle}\})]\ \wedge\ v = \mathsf{Ref}\, l} \quad \text{(VNEW)}$$

$$\frac{\Gamma \rhd e_1 : A_1 \qquad \Gamma \rhd e_2 : A_2}{\Gamma \rhd \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \lambda\, E\ h\ h'\ v\ p.\ \exists p'.\ \ p = \mathcal{R}^{\texttt{if}}_x(p') \wedge (E\langle x\rangle = \mathsf{true} \longrightarrow A_1\ E\ h\ h'\ v\ p') \wedge (E\langle x\rangle = \mathsf{false} \longrightarrow A_2\ E\ h\ h'\ v\ p') \wedge (E\langle x\rangle = \mathsf{true} \vee E\langle x\rangle = \mathsf{false})} \quad \text{(VIF)}$$

$$\frac{\Gamma \rhd e_1 : A \qquad \Gamma \rhd e_2 : B}{\Gamma \rhd \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \lambda\, E\ h\ h'\ v\ p.\ \exists\ p_1\ p_2\ h_1\ w.\ \ A\ E\ h\ h_1\ w\ p_1\ \wedge\ w \neq \bot \wedge B\ (E\langle x := w\rangle)\ h_1\ h'\ v\ p_2 \wedge p = \mathcal{R}^{\texttt{let}}_x(p_1, p_2)} \quad \text{(VLET)}$$

$$\frac{\Gamma \rhd e_1 : A \qquad \Gamma \rhd e_2 : B}{\Gamma \rhd e_1\ ;\ e_2 : \lambda\, E\ h\ h'\ v\ p.\ \exists\ p_1\ p_2\ h_1.\ \ A\ E\ h\ h_1 \perp p_1\ \wedge\ B\ E\ h_1\ h'\ v\ p_2 \wedge p = \mathcal{R}^{\texttt{comp}}(p_1, p_2)} \quad \text{(VCOMP)}$$

$$\frac{\Gamma \cup \{(\texttt{call}\, f, A)\} \rhd \mathit{body}_f : \Theta(A, f)}{\Gamma \rhd \texttt{call}\, f : A} \quad \text{(VCALL)}$$

$$\frac{\Gamma \cup \{(c.m(\overline{a}), A)\} \rhd \mathit{body}_{c,m} : \Phi(A, c, m, \overline{a})}{\Gamma \rhd c.m(\overline{a}) : A} \quad \text{(VSINV)}$$

$$\frac{\forall c.\ \Gamma \cup \{(x \cdot m(\overline{a}), A)\} \rhd \mathit{body}_{c,m} : \Psi(A, x, c, m, \overline{a})}{\Gamma \rhd x \cdot m(\overline{a}) : A} \quad \text{(VVINV)}$$

$$\frac{(e, A) \in \Gamma}{\Gamma \rhd e : A}\ \text{(VAX)} \qquad \frac{\Gamma \rhd e : A \quad \forall\ E\ h\ h'\ v\ p.\ A\ E\ h\ h'\ v\ p \longrightarrow B\ E\ h\ h'\ v\ p}{\Gamma \rhd e : B}\ \text{(VCONSEQ)}$$

The rules for function calls and method invocations make use of the following operators that model the effect of frame creation and the application of the resource algebra operations:

$$\begin{aligned}
\Theta(A, f) &= \lambda\, E\ h\ h'\ v\ p.\ A\ E\ h\ h'\ v\ (\mathcal{R}^{\texttt{call}}_f(p)) \\
\Phi(A, c, m, \overline{a}) &= \lambda\, E\ h\ h'\ v\ p. \\
&\qquad \forall\ E'.\ E = \mathit{Env}(\texttt{self} :: \mathit{pars}_{c,m}, \mathsf{null} :: \overline{a}, E') \longrightarrow A\ E'\ h\ h'\ v\ (\mathcal{R}^{\texttt{invs}}_{c,m,\overline{a}}(p)) \\
\Psi(A, x, c, m, \overline{a}) &= \lambda\, E\ h\ h'\ v\ p. \\
&\qquad \forall\ E'\ l.\ \ (E'\langle x\rangle = \mathsf{Ref}\, l \wedge h(l) = c\ \wedge E = \mathit{Env}(\texttt{self} :: \mathit{pars}_{c,m}, x :: \overline{a}, E')) \\
&\qquad\qquad \longrightarrow A\ E'\ h\ h'\ v\ (\mathcal{R}^{\texttt{invv}}_{x,m,\overline{a}}(p)).
\end{aligned}$$

The axioms (VNULL to VNEW) directly model the corresponding rules in the operational semantics, with constants for the resource tuples. The VIF rule uses the appropriate assertion based on the boolean value in the variable x. Since the evaluation of the branch condition does not modify the heap we only existentially quantify over the cost component p'. In contrast, rule VLET existentially quantifies over the intermediate result value w, the heap h_1 resulting

from evaluating e_1, and the resources from e_1 and e_2. Apart from the absence of environment update, rule VCOMP is similar to VLET.

The rules for recursive functions and methods generalize Hoare's original rule for parameter-less recursive procedures, which allows one to prove the body of a procedure satisfying an assertion under the assumption that recursive calls already satisfy that assertion. In our rules, the explicit context holds the assumptions about recursive invocations, and we prove a modified assertion for the body which accounts for the change in resources and environment when entering and exiting the function or procedure. The modifications match the corresponding operational rules CALL, SINV and VINV. In rule VCALL, the modification Θ only affects the resources, as the operational rule CALL leaves the environment, the heaps, and the result value untouched. In the case of VSINV and VVINV, the construction of a new frame in the operational rules corresponds to the universal quantification over the environment associated with the *caller*, E', in the definitions of operators Φ and Ψ. In both cases, the environment associated with the body, E, arises from this outer environment E' by the $Env(\_, \_, \_)$ function. Again, the costs of the method call are applied by requiring that the body satisfies an assertion whose resource component makes A true after the application of the appropriate operator from $\mathcal{R}$. Recursive calls are proved by projecting from the context using the rule VAX.

The consequence rule, VCONSEQ, derives an assertion B that follows by implication from another assertion A in the metalogic Isabelle/HOL; this is the place where all logical and arithmetic reasoning enters the system.

The rules for program composition, VLET and VCOMP, compose two assertions satisfied by the subexpressions. The composed assertion is formed by existential quantification over the intermediate state, and connecting it to the post state. The design issues behind these rules are discussed further next.

3.3. Discussion

In Hoare-style program logics, the purpose of auxiliary variables is to link pre- and post-conditions by "freezing" the values of (program or other) variables in the initial state, so that they can be referred to in the post-condition. Formally, auxiliary variables need to be universally quantified in the interpretation of judgements in order to treat variables of arbitrary domain, and their interaction with the rule of consequence. This quantification may either happen explicitly at the object level or implicitly, where pre- and post-condition are predicates over pairs of states and the domain of auxiliary variables. Kleymann [21] showed that by instantiating the domain of auxiliary variables to the category of states one may embed VDM-style program logics in Hoare-style logics. In particular, VDM's characteristic feature that allows the initial values of program variables to be referred to in the post-condition was modelled by giving both assertion components different types: preconditions are predicates on (initial) states while post-conditions are binary relations on states.

Kleymann's proposal to instantiate the domain of auxiliary variables to states also allows one to employ further quantification in the definition of assertions. Such additional quantification typically ranges over semantic entities rather than program variables. As an example, consider an object-orientated language of terminating commands c, and a predicate $classOf(s, x, A)$ that is satisfied if the object pointed to by variable x in state s is of class A. A Kleymann-style judgement

$$\{\lambda\ Z\ s.\ Z = s\}c\{\lambda\ Z\ t.\ \forall\ A.\ classOf(Z, x, A) \to classOf(t, x, A)\} \tag{1}$$

states that the class of object x remains unaffected by program c, with quantification occurring inside the post-condition. This idiom is preferable to the alternative where the domain of auxiliary variables is chosen to be the category of class names:

$$\{\lambda\ A\ s.\ classOf(s, x, A)\}c\{\lambda\ A\ t.\ classOf(t, x, A)\}. \tag{2}$$

Although this format appears simpler than the first, it has the disadvantage that the type of assertions conceptually depends on the property one wishes to prove — an immediate obstacle to a compositionality if other program fragments require different instantiations.

Our rules combine pre- and post-conditions and enforce Kleymann's discipline. Were we to adapt our assertion format to an imperative language, the above property would be expressed as:

$$\Gamma \rhd c : \lambda\ s\ t.\ \forall\ A.\ classOf(s, x, A) \to classOf(t, x, A),$$

which uses the same inner quantification as (1) but avoids the auxiliary variable Z, similarly to a formulation in VDM.

Obviously, since there are no auxiliary variables used in our assertion format, there is no need in our proof system for rules that introduce or eliminate quantifiers over these variables (as are used elsewhere, e.g. [22]). The formal completeness result to follow establishes that the present system suffices for deriving arbitrary valid statements, so we are clearly expressive enough. And pragmatically, a proof system that is mostly syntax-directed supports the automated generation of verification conditions better than a system with rules for introducing auxiliary variables at arbitrary points.[6]

Another difference to Hoare's calculus occurs in VLET. This rule combines rules for variable assignment and sequential program composition (the latter one corresponding more closely to rule VCOMP), motivated by the syntactic structure of Grail and its environment-based operational semantics. In contrast to Hoare's assignment rule

$$\frac{}{\{A[e/x]\}x := e\{A\}}$$

Our rule VLET does not involve any syntactic manipulation of formulae. Instead, the update of the environment (in an imperative setting: the assignment to the store) is modelled in exactly the same way as in the operational semantics, avoiding the (at first sight) counter-intuitive syntactic substitution in Hoare's rule.

Our composition rules admit traversals in a forward direction, combining arbitrary assertions for subexpressions. In contrast, in Hoare's rule for program composition

$$\frac{\{A\}c_1\{B\} \quad \{B\}c_2\{C\}}{\{A\}c_1; c_2\{C\}}$$

one must find a common intermediate *assertion*, perhaps using consequence; instead, we quantify only over intermediate *states* within a compound assertion. Again, our rule is more closely related to the corresponding rule in VDM:

$$\frac{\{A\}c_1\{\lambda\, s\ t.\ B\ t \wedge C\ s\ t\} \quad \{B\}c_2\{D\}}{\{A\}c_1; c_2\{D \circ C\}},$$

which uses relational composition to combine post conditions.

3.4. Admissible rules

In addition to the proof rules given Section 3.2, we need some rules to simplify reasoning about concrete programs, and some others to help establish completeness. All of these rules involve the context of assumptions.

First, we introduce the concept of *specification tables*. These associate an assertion to each function name or method invocation.

Definition 4. A *specification table ST* consists of functions of type $\mathcal{F} \to \mathcal{A}$, $\mathcal{C} \to \mathcal{M} \to \overline{args} \to \mathcal{A}$, and $\mathcal{X} \to \mathcal{M} \to \overline{args} \to \mathcal{A}$, where $\overline{args}$ is the type of argument lists. We write $ST(f)$, $ST(c, m, \overline{a})$ and $ST(x, m, \overline{a})$ for the respective access operations.

The additional rules are as follows:

$$\frac{\Gamma \rhd e : A}{\Gamma \cup \Delta \rhd e : A} \quad \text{(VWEAK)}$$

$$\frac{\Gamma \rhd e : A \quad \forall\, d\ B.\ (d, B) \in \Gamma \longrightarrow \Delta \rhd d : B}{\Delta \rhd e : A} \quad \text{(VCTXT)}$$

$$\frac{\{(d, B)\} \cup \Gamma \rhd e : A \quad \Gamma \rhd d : B}{\Gamma \rhd e : A} \quad \text{(VCUT)}$$

$$\frac{\Gamma \models ST \quad (e, A) \in \Gamma}{\emptyset \rhd e : A} \quad \text{(MUTREC)}$$

[6] Nonetheless, it is still desirable to make use of some richer admissible rules to help build up proofs for complete programs; these are introduced in Section 3.4.

$$\frac{\Gamma \models ST \quad (c.m(\overline{a}), ST(c, m, \overline{a})) \in \Gamma}{\emptyset \rhd c.m(\overline{b}) : ST(c, m, \overline{b})} \quad (\textsc{AdaptS})$$

$$\frac{\Gamma \models ST \quad (x \cdot m(\overline{a}), ST(x, m, \overline{a})) \in \Gamma}{\emptyset \rhd y \cdot m(\overline{b}) : ST(y, m, \overline{b})} \quad (\textsc{AdaptV})$$

The first two rules, VWEAK and VCTXT, are proven by an induction on the derivation of $\Gamma \rhd e : A$. The further cut rule, VCUT, follows easily from VCTXT. The cut rules eliminate the need for introducing a second form of judgement used previously in the literature (e.g., [23]) when establishing soundness of the rule MUTREC for mutually recursive program fragments. This proof will now be outlined. Contexts whose entries arise uniformly from specification tables are of particular interest.

Definition 5. The context Γ *respects* specification table ST, notation $\Gamma \models ST$, if all $(e, A) \in \Gamma$ satisfy one of the three following conditions

- $(e, A) = (\texttt{call}\, f, ST(f))$ for some f with $\Gamma \rhd body_f : \Theta(ST(f), f)$
- $(e, A) = (c.m(\overline{a}), ST(c, m, \overline{a}))$ for some c, m and $\overline{a}$, and all $\overline{b}$ satisfy

  $$\Gamma \rhd body_{c,m} : \Phi(ST(c, m, \overline{b}), c, m, \overline{b}),$$

- $(e, A) = (x \cdot m(\overline{a}), ST(x, m, \overline{a}))$ for some x, m and $\overline{a}$, and all c, y, and $\overline{b}$ satisfy

  $$\Gamma \rhd body_{c,m} : \Psi(ST(y, m, \overline{b}), y, c, m, \overline{b}).$$

Here, the operators Θ, Φ, and Ψ are those defined in Section 3.2. Using rule VCUT, is not difficult to prove that this property is closed under subcontexts.

Lemma 6. *If* $(e, A) \cup \Gamma \models ST$ *then* $\Gamma \models ST$.

Based on Lemma 6, rule MUTREC can be proved by induction on the size of Γ. Notice that the conclusion relates e to A in the empty proof context — and thus, by rule VWEAK, in *any* context.

The remaining admissible rules ADAPTS and ADAPTV amount to variations of MUTREC for method invocations. In these rules, the expression in the conclusion may differ (be adapted) from the expression stored in the context, as long as this difference occurs only in the method arguments (including the object on which a virtual method is invoked) and is reflected in the assertions. In Hoare logics, the adaptation of method specifications is related to the adaptation of auxiliary variables, a historically tricky issue in the formal understandings of program logics [24,23,21]. For example, Nipkow [23] adapts auxiliary variables in the rule of consequence; this allows him to adapt them also when accessing method specifications from contexts. In addition to admitting such an adaptation using universal quantification in the definition of method specifications (see the discussion in the previous section), our rules also allow differences in *syntactic* components, namely the method arguments.

In order to show ADAPTS sound, we first prove

Lemma 7. *If* $\Gamma \models ST$ *and* $(c.m(\overline{a}), ST(c, m, \overline{a})) \in \Gamma$ *then* $\Gamma \setminus (c.m(\overline{a}), ST(c, m, \overline{a})) \rhd c.m(\overline{b}) : ST(c, m, \overline{b})$

using rule VCUT. The conclusion in this lemma already involves method arguments, namely $\overline{b}$, that may be different from the arguments used in the context $\overline{a}$. From this, rule ADAPTS follows by repeated application of Lemma 6. The proof of rule ADAPTV is similar.

3.5. Verification examples

Before treating the metalogical issues of soundness and completeness, we describe a number of verification examples that we have carried out using our implementation of the program logic in Isabelle/HOL. These examples show how to use the argument adaptation rules for method invocations, and how to specify and verify properties of resource consumption using resource algebras. All proofs follow a similar strategy which we outline first.

3.5.1. Proof strategy

We will always aim to verify that some methods[7] $c_1.m_1, \ldots, c_n.m_n$ with bodies $e_1, \ldots, e_n$, respectively, satisfy their specifications in ST, i.e. that

$$\emptyset \rhd c_i.m_i(\overline{a}_i) : ST(c_i, m_i, \overline{a}_i) \tag{3}$$

holds for each i and arbitrary $\overline{a}_i$. In each case, the proof consists of defining a single context Γ that contains entries for all methods $c_i.m_i$ and satisfies $\Gamma \models ST$, and then applying rule ADAPTS separately for each i. A suitable context is one that contains

- at least all entries $(c.m(\overline{a}), ST(c, m, \overline{a}))$ where $c.m(\overline{a})$ occurs in at least one body e_i (we may discard the entries for methods that have been verified previously), and
- at least one entry $(c_i.m_i(\overline{a}_i), ST(c_i, m_i, \overline{a}_i))$ for each $i \in \{1, \ldots, n\}$, where the $\overline{a}_i$ are distinct meta-variables.

In order to establish $\Gamma \models ST$, we then prove

$$\forall \overline{b}.\ \Gamma \rhd body_{c,m} : \Phi(ST(c, m, \overline{b}), c, m, \overline{b})$$

for each pair (c, m) for which there exists an $\overline{a}$ with $(c.m(\overline{a}), ST(c, m, \overline{a})) \in \Gamma$. Here, the operator Φ is the one defined in Section 3.2. In the proofs of these lemmas, all method invocations for which there is an entry in Γ are verified using the axiom rule VAX, while invocations of methods for which property (3) has been established previously are discharged by rule VWEAK. Together, these lemmas yield a verification of $\Gamma \models ST$ in which each method body has been verified only once.

3.5.2. Datatype representation predicates

To verify code resulting from compiling algebraic datatypes, we employ representation predicates which describe when a JVM (reference) value represents a member of the abstract type. For the verification examples we discuss in the present paper, "lightweight" predicates will suffice, only specifying the size of a value and the heap region it inhabits, but not the abstract value itself. The definition of these predicates reflects the layout chosen by our compiler and models representations without internal sharing. For example, consider the type of integer lists, and the layout for such lists given in Section 2. The corresponding representation predicate $h \models_{list(n,X)} v$ mandates that the heap h contains a well laid-out (non-overlapping) list of length n starting at value v and occupying locations X:

$$\frac{h(l) = \mathsf{LIST} \quad h(l).\mathsf{TAG} = 1}{h \models_{list(0,\{l\})} \mathsf{Ref}\, l} \qquad \frac{\begin{array}{ll} h(l) = \mathsf{LIST} & l \notin X \\ h(l).\mathsf{TAG} = 2 & h \models_{list(n,X)} h(l).\mathsf{TL} \end{array}}{h \models_{list(n+1,X\cup\{l\})} \mathsf{Ref}\, l}\ .$$

3.5.3. Append

Our first example is for the method LIST.append depicted in Fig. 1. The program proceeds by induction on the first argument, but modifies only the second argument. It also allocates fresh memory when constructing the result. The result overlaps only with the second argument, and the content of all locations in the initial heap remains unchanged. Formally, this understanding may be expressed by the specification table entry

$$\begin{array}{l} ST(\mathsf{LIST}, \mathsf{append}, \overline{a}) = \\ \lambda\, E\, h\, h'\, v\, p.\, \forall\, Y\, n\, m. \\ \left(\begin{array}{l} (\exists\, X\, x\, y. \overline{a} = [\mathtt{var}\ x, \mathtt{var}\ y] \wedge h \models_{list(n,X)} E\langle x\rangle \wedge h \models_{list(m,Y)} E\langle y\rangle \wedge X \cap Y = \emptyset) \\ \longrightarrow (\exists\, Z.\ h' \models_{list(n+m,Z)} v \wedge Z \cap dom\, h = Y \wedge h =_{dom\, h} h'). \end{array}\right) \end{array}$$

It asserts that the result v represents a list of length $n + m$ in the final heap, provided that the arguments represent non-overlapping lists of length n and m in the initial heap, respectively. Furthermore, the specification guarantees that the region inhabited by the result may at most overlap with the region inhabited by the second argument, and that the

[7] To simplify the explanation, we outline the verification strategy only for static methods.

content of all locations present in the initial heap remains unchanged. The last two conditions restrict which locations may be affected by the execution of the program and are necessary for a modular verification.

We verify that

$$\forall \overline{a}.\ \emptyset \rhd \mathsf{LIST.append}(\overline{a}) : ST(\mathsf{LIST}, \mathsf{append}, \overline{a}) \tag{4}$$

holds following the strategy outlined above.

In the first step, a suitable context is given by

$$\Gamma_{\mathsf{append}} = \{(\mathsf{LIST.append}([\mathtt{var\ t}, \mathtt{var\ l_2}]), ST(\mathsf{LIST}, \mathsf{append}, [\mathtt{var\ t}, \mathtt{var\ l_2}]))\},$$

since the only invocation in the body of append is LIST.append([var t, var $\mathtt{l}_2$]). The second step consists of proving the lemma

$$\forall \overline{b}.\ \Gamma_{\mathsf{append}} \rhd body_{\mathsf{LIST,append}} : \Phi(ST(\mathsf{LIST}, \mathsf{append}, \overline{b}), \mathsf{LIST}, \mathsf{append}, \overline{b}). \tag{5}$$

This is achieved by automatically applying the syntax-directed proof rules and using rule VAX at the invocation of LIST.append([var t, var $\mathtt{l}_2$]). Two verification conditions remain, one for each outcome of the conditional. The condition for Nil is

$$\begin{pmatrix} l \in dom\ h \wedge h(l).\mathsf{TAG} = 1 \wedge E'\langle x\rangle = \mathsf{Ref}\ l \wedge h \models_{list(N,X)} \mathsf{Ref}\ l \wedge \\ E'\langle y\rangle = \mathsf{Ref}\, l_2 \wedge h \models_{list(M,Y)} \mathsf{Ref}\ l_2 \wedge X \cap Y = \emptyset \end{pmatrix} \\ \longrightarrow\ \exists\ Z.\ h \models_{list(N+M,Z)} \mathsf{Ref}\ l_2 \wedge Z \cap dom\ h = Y.$$

Note that all datatype predicates refer to h as no heap modification is performed by the program. It is easy to see that $h(l).\mathsf{TAG} = 1 \wedge h \models_{list(N,X)} \mathsf{Ref}\ l$ implies $N = 0$ and $X = \{l\}$, so the goal follows by instantiating Z to Y. In the verification condition for the Cons-case, we let the predicate $A(h, l, l_2, r, h_1)$ abbreviate

$$\forall\ N\ M\ l_1\ X\ Y. \\ \begin{pmatrix} (h(l).\mathsf{TL} = \mathsf{Ref}\, l_1 \wedge h \models_{list(N,X)} \mathsf{Ref}\ l_1\ \wedge h \models_{list(M,Y)} \mathsf{Ref}\ l_2 \wedge X \cap Y = \emptyset) \\ \longrightarrow (\exists\ V\ l'.\ r = \mathsf{Ref}\, l' \wedge h_1 \models_{list(N+M,V)} \mathsf{Ref}\, l'\ \wedge V \cap dom\ h = Y \wedge h =_{dom\ h} h_1) \end{pmatrix}.$$

This term represents the induction hypothesis, i.e. the specification of the recursive call to append. The verification condition may then be written as

$$\begin{pmatrix} l \in dom\ h \wedge h(l).\mathsf{TAG} = 2 \wedge E'\langle x\rangle = \mathsf{Ref}\, l \wedge h \models_{List(N,X)} \mathsf{Ref}\, l \wedge \\ E'\langle y\rangle = \mathsf{Ref}\, l_2 \wedge h \models_{List(M,Y)} \mathsf{Ref}\, l_2 \wedge X \cap Y = \emptyset\ \wedge\ A(h, l, l_2, r, h_1) \end{pmatrix} \\ \longrightarrow \exists\ Z.\ h' \models_{List(N+M,Z)} \mathsf{Ref}\ freshloc(dom\ h_1) \wedge Z \cap dom\ h = Y \wedge h =_{dom\ h} h'.$$

From $h(l).\mathsf{TAG} = 2$ we obtain that there are K, l_3 and U such that $N = K + 1$, $h(l).\mathsf{TL} = \mathsf{Ref}\, l_3$, $U \cup \{l_3\} = X$, $l_3 \notin U$ and $h \models_{List(K,U)} \mathsf{Ref}\, l_3$. We can thus apply the induction hypothesis, instantiating N with K and X with U and obtain $h_1 \models_{list(N+M,V)} \mathsf{Ref}\, l'$, $Z \cap dom\ h = Y$ and $h =_{dom\ h} h_1$ for some V and l'. The right-hand side of the verification condition may then be proven by instantiating Z by $V \cup freshloc(dom\ h_1)$.

The proof of the verification conditions involves instantiation of quantifiers, plus elimination, introduction and preservation lemmas for datatype predicates such as

$$(h \models_{list(n,X)} v \wedge h =_X h') \longrightarrow h' \models_{list(n,X)} v$$

which are themselves proved by induction on $h \models_{list(n,X)} v$. (In the absence of automatically derived predicates and lemmas for datatype definitions, the proof of our verification conditions relies on manual proof of required properties for a fixed set of datatypes.)

As explained in the general description of our verification approach, the result (5) now immediately yields $\Gamma_{\mathsf{append}} \models ST$ and the correctness statement (4) follows using rule ADAPTS.

The specification of append may be refined to include quantitative aspects using appropriate resource algebras. For example, we can verify that all four metrics that make up $\mathcal{R}^{\mathsf{Count}}$ depend linearly on the length of the list represented by the first argument, and so does the memory consumption. To this end we consider a modified specification

$$
\begin{array}{l}
\texttt{method}\ \mathsf{LIST\ TREE.flatten(t)} = \texttt{call}\ f \\
\left[\begin{array}{lcl}
f & \mapsto & \texttt{let}\ \mathsf{tg} = \mathsf{t.TAG}\ \texttt{in}\ \texttt{let}\ \mathsf{b} = \mathit{iszero}\ \mathsf{tg}\ \texttt{in} \\
 & & \texttt{if}\ \mathsf{b}\ \texttt{then call}\ f_0\ \texttt{else call}\ f_1 \\
f_0 & \mapsto & \texttt{let}\ \mathsf{i} = \mathsf{t.CONT}\ \texttt{in}\ \texttt{let}\ \mathsf{tg} = \texttt{imm}\ 1\ \texttt{in} \\
 & & \texttt{let}\ \mathsf{t} = \texttt{new}\ \mathsf{LIST}\ [\mathsf{TAG} := \mathsf{tg}]\ \texttt{in}\ \texttt{let}\ \mathsf{tg} = \texttt{imm}\ 2\ \texttt{in} \\
 & & \texttt{new}\ \mathsf{LIST}\ [\mathsf{TAG} := \mathsf{tg};\ \mathsf{HD} := \mathsf{i};\ \mathsf{TL} := \mathsf{t}] \\
f_1 & \mapsto & \texttt{let}\ \mathsf{l} = \mathsf{t.LEFT}\ \texttt{in}\ \texttt{let}\ \mathsf{r} = \mathsf{t.RIGHT}\ \texttt{in}\ \texttt{let}\ \mathsf{l} = \mathsf{TREE.flatten}(\texttt{var}\ \mathsf{l})\ \texttt{in} \\
 & & \texttt{let}\ \mathsf{r} = \mathsf{TREE.flatten}(\texttt{var}\ \mathsf{r})\ \texttt{in}\ \mathsf{LIST.append}([\texttt{var}\ \mathsf{l}, \texttt{var}\ \mathsf{t}])
\end{array}\right]
\end{array}
$$

Fig. 3. Code of method flatten.

which imposes linear constraints on these costs using symbolic factors $AppTimeF, \ldots, AppHeapF$ and constants $AppTimeC, \ldots, AppHeapC$.

$$
\begin{array}{l}
ST(\mathsf{LIST}, \mathsf{append}, \overline{a}) = \lambda\, E\, h\, h'\, v\, p. \\
\quad \forall\, Y\, n\, m.\ \left(\begin{array}{l}\exists\, X\, x\, y.\quad \overline{a} = [\texttt{var}\ x, \texttt{var}\ y] \wedge h \models_{list(n,X)} E\langle x\rangle \wedge \\ \qquad\qquad\ \ h \models_{list(m,Y)} E\langle y\rangle \ \wedge\ X \cap Y = \emptyset\end{array}\right) \longrightarrow \\
\qquad \left(\begin{array}{l}\exists\, Z.\quad h' \models_{list(n+m,Z)} v \ \wedge\ Z \cap dom\, h = Y \ \wedge\ h =_{dom\, h} h' \wedge \\ \qquad p = \langle\quad (AppTimeF * n + AppTimeC) \\ \qquad\qquad\quad (AppCallF * n + AppCallC) \\ \qquad\qquad\quad (AppInvF * n + AppInvC) \\ \qquad\qquad\quad (AppStackF * n + AppStackC)\quad \rangle \wedge \\ \qquad |dom\, h'| = |dom\, h| + AppHeapF * n + AppHeapC\end{array}\right).
\end{array}
$$

Following the same verification process as before, we obtain verification conditions that, in addition to the assertions discussed above, include recurrence relations over the symbolic terms. The following numbers were obtained by solving these constraints manually

$AppTimeF$	35	$AppTimeC$	14
$AppCallF$	2	$AppCallC$	1
$AppInvF$	1	$AppInvC$	1
$AppStackF$	1	$AppStackC$	1
$AppHeapF$	1	$AppHeapC$	0

and indeed these values allow one to complete the formal proof.

3.5.4. Flatten

To continue with another example program, Fig. 3 shows the definition of a method that flattens a binary tree into a list. The code was obtained by pretty-printing the output of our compiler for a Camelot file that extends the earlier code for append by

```
type itree = Leaf of int | Node of itree * itree

let flatten t = match t with
                  Leaf(i) => Cons(i,Nil)
                | Node(l,r) => append (flatten l) (flatten r).
```

Trees are laid out in memory as objects of class TREE, where leafs (TAG $= 0$) contain a field CONT and nodes (TAG $\neq 0$) contain subtrees LEFT and RIGHT.

Similarly to the previous example, we define a specification table entry for flatten,

$$\begin{array}{l} ST(\mathsf{TREE}, \mathsf{flatten}, \overline{a}) = \lambda\, E\, h\, h'\, v\, p. \\ \quad \forall\, n\, x.\ \ (\exists\, X.\, \overline{a} = [\mathtt{var}\ x] \ \wedge\ h \models_{tree(n,X)} E\langle x\rangle) \longrightarrow \\ \qquad\qquad (\exists\, Z.\, h' \models_{list(2^n,Z)} v \ \wedge\ Z \cap dom\, h = \emptyset \ \wedge\ h =_{dom\, h} h') \end{array}$$

which universally quantifies over the argument name x. It asserts that the result v represents a list of length 2^n in the final heap, provided that the argument represents a balanced binary tree of height n in the initial heap. Moreover, the region inhabited by the result, Z, does not overlap with h (i.e. the list is represented in freshly allocated memory), and the content of all locations in $dom\, h$ remains unchanged. The datatype representation predicate $h \models_{tree(n,X)} v$ is defined in a similar way to the list predicate $h \models_{list(n,X)} v$, namely:

$$\frac{h(l) = \mathsf{TREE} \quad h(l).\mathsf{TAG} = 0}{h \models_{tree(0,\{l\})} \mathsf{Ref}\, l} \qquad \frac{\begin{array}{ll} h(l) = \mathsf{TREE} & l \notin L \cup R \\ h(l).\mathsf{TAG} \neq 0 & h \models_{tree(n,L)} h(l).\mathsf{LEFT} \\ L \cap R = \emptyset & h \models_{tree(n,R)} h(l).\mathsf{RIGHT} \end{array}}{h \models_{tree(n+1,L\cup R\cup\{l\})} \mathsf{Ref}\, l}\ .$$

Once more, following the prescribed verification strategy, we prove

$$\forall \overline{a}.\ \emptyset \rhd \mathsf{TREE.flatten}(\overline{a}) : ST(\mathsf{TREE}, \mathsf{flatten}, \overline{a}). \tag{6}$$

The context defined in the first step may be chosen as

$$\Gamma_{\mathsf{flatten}} = \left\{ \begin{array}{l} (\mathsf{TREE.flatten}([\mathtt{var}\ \mathsf{v}_2]), ST(\mathsf{TREE}, \mathsf{flatten}, [\mathtt{var}\ \mathsf{v}_2])), \\ (\mathsf{TREE.flatten}([\mathtt{var}\ \mathsf{v}_3]), ST(\mathsf{TREE}, \mathsf{flatten}, [\mathtt{var}\ \mathsf{v}_3])) \end{array} \right\}$$

since the method append has already been verified.

In the verification of

$$\forall \overline{b}.\ \Gamma_{\mathsf{flatten}} \rhd body_{\mathsf{TREE,flatten}} :\ \Phi(ST(\mathsf{TREE}, \mathsf{flatten}, \overline{b}), \mathsf{TREE}, \mathsf{flatten}, \overline{b}), \tag{7}$$

the two invocations of flatten are discharged by rule VAX, while the invocation of append is discharged by appealing to property (4) using VWEAK. Once more, the proofs of the verification conditions involve case analysis on the datatype representation predicates, the instantiation of quantifiers, and the application of datatype preservation lemmas for trees and lists. The preconditions of the latter are satisfied thanks to the separation conditions in the specifications of append and flatten.

Similarly to the verification of append, we obtain $\Gamma_{\mathsf{flatten}} \models ST$ from the result (7), and the correctness of flatten, i.e property (6), follows using rule ADAPTS.

To verify resource consumption for this method using $\mathcal{R}^{\mathsf{Count}}$, we observe that the costs of flatten depend on those of append, plus the costs of two recursive invocations of flatten on subtrees. The resulting recurrences for the four additive metrics follow a uniform pattern *FlCost* of type $(\mathcal{N} \Rightarrow \mathcal{N}) \Rightarrow \mathcal{N} \Rightarrow \mathcal{N} \Rightarrow \mathcal{N} \Rightarrow \mathcal{N}$, defined by

$$\begin{array}{l} FlCost\ app\ base\ step\ 0 = base \\ FlCost\ app\ base\ step\ (Suc\ n) = step + app(2^n) + 2 * (FlCost\ app\ base\ step\ n). \end{array}$$

This functional describes how the costs of flattening a tree depend on the height n and the costs of append. The costs of flattening a tree of height zero are given by the parameter *base*. The costs of flattening a tree of height $n+1$ are given by the costs of two flattening of a tree of height n, the costs of one invocation of append on a list of length 2^n, and the costs resulting from the remaining instructions in the loop body (parameter *step*). Note that this pattern is modular in the costs of append.

The costs with respect to the four additive metrics may now be obtained by instantiating *FlCost* with the appropriate parameters. The parameters *base* and *step* are specific for the implementation of flatten and were once again obtained experimentally. The append-specific costs are given by the linear terms proven earlier, using the factors and constants

$$\begin{array}{lll}
\texttt{method}\ \mathsf{BOOL}\ \mathsf{EO.even(x)} & = & \texttt{let}\ \mathsf{b} = \mathit{iszero}\ \mathsf{x}\ \texttt{in}\ \texttt{let}\ \mathsf{x} = \mathsf{x} - 1\ \texttt{in} \\
 & & \texttt{if}\ \mathsf{b}\ \texttt{then}\ \texttt{imm}\ \texttt{true}\ \texttt{else}\ \mathsf{EO.odd}([\texttt{var}\ \mathsf{x}]) \\
\texttt{method}\ \mathsf{BOOL}\ \mathsf{EO.odd(y)} & = & \texttt{let}\ \mathsf{b} = \mathit{iszero}\ \mathsf{y}\ \texttt{in}\ \texttt{let}\ \mathsf{y} = \mathsf{y} - 1\ \texttt{in} \\
 & & \texttt{if}\ \mathsf{b}\ \texttt{then}\ \texttt{imm}\ \texttt{false}\ \texttt{else}\ \mathsf{EO.even}([\texttt{var}\ \mathsf{y}])
\end{array}$$

Fig. 4. Code of methods even and odd.

AppTimeF, ..., *AppHeapC*, although the costs for a different implementation of append could be substituted easily.

$$\begin{array}{l}
\mathit{FlTime}\ n = \mathit{FlCost}\ (\lambda\, n.\, \mathit{AppTimeF} * n + \mathit{AppTimeC})\ 38\ 22\ n \\
\mathit{FlCall}\ n = \mathit{FlCost}\ (\lambda\, n.\, \mathit{AppCallF} * n + \mathit{AppCallC})\ 2\ 2\ n \\
\mathit{FlInv}\ n = \mathit{FlCost}\ (\lambda\, n.\, \mathit{AppInvF} * n + \mathit{AppInvC})\ 1\ 1\ n \\
\mathit{FlHeap}\ n = \mathit{FlCost}\ (\lambda\, n.\, \mathit{AppHeapF} * n + \mathit{AppHeapC})\ 2\ 0\ n.
\end{array}$$

The recurrence equation for the frame stack height is given by

$$\begin{array}{l}
\mathit{FlStack}\ 0 = 1 \\
\mathit{FlStack}\ (\mathit{Suc}\ n) = 1\ +\ \mathit{max}(\mathit{FlStack}\ n, 2^n + 1).
\end{array}$$

The verification of the extended specification

$$\begin{array}{l}
\mathit{ST}(\mathsf{TREE}, \mathsf{flatten}, \overline{a}) = \lambda\, E\, h\, h'\, v\, p. \\
\quad \forall\, n\, x.\ \ (\exists\, X.\, \overline{a} = [\texttt{var}\ x]\ \wedge\ h \models_{\mathit{tree}(n,X)} E\langle x\rangle) \longrightarrow \\
\qquad \left(\begin{array}{ll}
\exists\, Z. & h' \models_{\mathit{list}(2^n,Z)} v\ \wedge\ Z \cap \mathit{dom}\, h = \emptyset\ \wedge\ h =_{\mathit{dom}\, h} h'\ \wedge \\
 & p = \langle(\mathit{FlTime}\ n)\ (\mathit{FlCall}\ n)\ (\mathit{FlInv}\ n)\ (\mathit{FlStack}\ n)\rangle\ \wedge \\
 & |\mathit{dom}\, h'| = |\mathit{dom}\, h| + \mathit{FlHeap}\ n
\end{array}\right)
\end{array}$$

again proceeds following the same structure as for the simpler specification (6). As expected, unfolding the recurrence equations leads to functions of exponential growth, since the index n in the predicate $h \models_{\mathit{tree}(n,X)} v$ denotes the height of the tree.

3.5.5. Even/odd

As a third example, we verify that a run of the mutually recursive methods even and odd (code in Fig. 4) exhibits the expected alternation of method invocations. This demonstrates that our rule MUTREC indeed works for mutually recursive methods, and also shows how we may use resource algebras for reasoning about execution traces. In particular, we consider the resource algebra $\mathcal{R}^{\mathsf{InvTr}}$ that collects the static method invocations in execution order. We first define the auxiliary functions $f_{\mathit{Even}}, f_{\mathit{Odd}} : \mathcal{N} \to \overline{\mathit{expr}}$ that calculate the list of invocations occurring inside a method call, relative to a given input.

$$\begin{array}{lll}
f_{\mathit{Even}}\ 0 = [\,] & & \\
f_{\mathit{Even}}\ (\mathit{Suc}\ n) & = & \mathit{if}\ (\exists\, k.\, \mathit{Suc}\ n = 2 * k)\ \mathit{then}\ (f_{\mathit{Even}}\ n)\ @\ [\mathsf{EO.even}([\texttt{var}\ \mathsf{y}])] \\
 & & \mathit{else}\ (f_{\mathit{Even}}\ n)\ @\ [\mathsf{EO.odd}([\texttt{var}\ \mathsf{x}])] \\
f_{\mathit{Odd}}\ 0 = [\,] & & \\
f_{\mathit{Odd}}\ (\mathit{Suc}\ n) & = & \mathit{if}\ (\exists\, k.\, \mathit{Suc}\ n = 2 * k)\ \mathit{then}\ (f_{\mathit{Odd}}\ n)\ @\ [\mathsf{EO.odd}([\texttt{var}\ \mathsf{x}])] \\
 & & \mathit{else}\ (f_{\mathit{Odd}}\ n)\ @\ [\mathsf{EO.even}([\texttt{var}\ \mathsf{y}])].
\end{array}$$

Next, the specification table entries for even and odd are defined. The specifications

$$\begin{array}{l}
\mathit{ST}(\mathsf{EO}, \mathsf{even}, \overline{a})\ =\ \lambda\, E\, h\, h'\, v\, p. \\
\quad \forall\, w.\ \ (\overline{a} = [\texttt{var}\ w]\ \wedge\ E\langle w\rangle \geq 0) \longrightarrow \left(\begin{array}{l} v = \mathit{is_even}(E\langle w\rangle)\ \wedge \\ p = \mathsf{EO.even}([\texttt{var}\ w]) :: (f_{\mathit{Even}}(E\langle w\rangle)) \end{array}\right) \\
\mathit{ST}(\mathsf{EO}, \mathsf{odd}, \overline{a})\ =\ \lambda\, E\, h\, h'\, v\, p. \\
\quad \forall\, w.\ \ (\overline{a} = [\texttt{var}\ w]\ \wedge\ E\langle w\rangle \geq 0) \longrightarrow \left(\begin{array}{l} v = \mathit{is_odd}(E\langle w\rangle)\ \wedge \\ p = \mathsf{EO.odd}([\texttt{var}\ w]) :: (f_{\mathit{Odd}}(E\langle w\rangle)) \end{array}\right)
\end{array}$$

$$\left[\begin{array}{lll} f & \mapsto & \texttt{let}\ n = n{+}1\ \texttt{in}\ \texttt{let}\ l = \texttt{new}\ \mathsf{LIST}\ [\mathsf{HD} := F(n,l), \mathsf{TL} := l]\ \texttt{in} \\ & & \mathsf{C.M}(\texttt{var}\ n, \texttt{var}\ l)\ ;\ \texttt{let}\ b = \mathit{if}\ n < 42\ \mathit{then}\ \texttt{imm true}\ \mathit{else}\ \texttt{imm false}\ \texttt{in} \\ & & \texttt{if}\ b\ \texttt{then call}\ f\ \texttt{else var}\ l \end{array}\right]$$

Fig. 5. A loop repeatedly invoking method C.M.

constrain the form of input arguments, prefix the outermost call to the list of internal invocations, and specify the result using the semantic functions

$$\mathit{is\_even}\ v = \mathit{if}\ (\exists\, n.\ v = 2 * n)\ \mathit{then}\ \texttt{true}\ \mathit{else}\ \texttt{false}$$
$$\mathit{is\_odd}\ v = \mathit{if}\ (\exists\, n.\ v = 2 * n)\ \mathit{then}\ \texttt{false}\ \mathit{else}\ \texttt{true}.$$

The context Γ_{EO} contains one entry for each method:

$$\Gamma_{\mathsf{EO}} = \left\{\begin{array}{l} (\mathsf{EO.even}([\texttt{var y}]), ST(\mathsf{EO}, \mathsf{even}, [\texttt{var y}])), \\ (\mathsf{EO.odd}([\texttt{var x}]), ST(\mathsf{EO}, \mathsf{odd}, [\texttt{var x}])) \end{array}\right\}.$$

The verification of $\Gamma_{\mathsf{EO}} \models ST$ unfolds each method body once, resolving the invocation of the opposite method by rule VAX. We thus obtain

$$\emptyset \rhd \mathsf{EO.even}([\texttt{var z}]) : ST(\mathsf{EO}, \mathsf{even}, [\texttt{var z}])$$

and

$$\emptyset \rhd \mathsf{EO.odd}([\texttt{var z}]) : ST(\mathsf{EO}, \mathsf{odd}, [\texttt{var z}])$$

for an arbitrary variable z using rule ADAPTS, and unfolding the definitions of f_{Even} and f_{Odd} yields results such as

$$\begin{array}{l} \emptyset \rhd \mathsf{EO.even}([\texttt{var z}]) : \lambda\, E\, h\, h'\, v\, p.\ \ E\langle \mathsf{z}\rangle = 4 \longrightarrow \\ \qquad\qquad\qquad\qquad\qquad\qquad v = \texttt{true} \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad p = [\mathsf{EO.even}([\texttt{var z}]), \mathsf{EO.odd}([\texttt{var x}]), \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{EO.even}([\texttt{var y}]), \mathsf{EO.odd}([\texttt{var x}]), \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{EO.even}([\texttt{var y}])]. \end{array}$$

3.5.6. Limits on parameter values

As a final example, we use the resource algebra $\mathcal{R}^{\mathsf{PVal}}$ to impose limits on the arguments of invocations of a (native) method. The code fragment in Fig. 5 shows a loop that repeatedly prepends a new element to a list l and then invokes C.M with l. In each iteration, the first argument, n, is intended to describe an upper bound on the length of l. The semantic function F is an arbitrary function which constructs a value for the new HD field, maybe depending on the current counter and tail (this is just suggestive and doesn't play a role in the example).

A policy that describes the desired relation is

$$P(\overline{a}) \equiv \exists\, x\, y\, n\ R.\ \overline{a} = [\texttt{var}\ x, \texttt{var}\ y]\ \wedge\ h \models_{\mathit{list}(n,R)} E\langle y\rangle\ \wedge\ n \leq E\langle x\rangle\ \wedge\ E\langle x\rangle \leq 42.$$

This expresses that the method must always be invoked on two arguments, the first one being a bound on the length of the list contained in the second; the list also has a fixed concrete limit on its length.

We can verify that all invocations of C.M arising from a call to f respect the policy, subject to conditions on the initial state. The verification of this:

$$\emptyset \rhd \texttt{call}\ f : \lambda\, E\, h\, h'\, v\, p.\ \ (\exists\, m\ R.\ h \models_{\mathit{list}(m,R)} E\langle l\rangle\ \wedge\ m \leq E\langle n\rangle) \longrightarrow p = (E\langle n\rangle < 42)$$

proceeds very similarly to the proofs described earlier.

3.6. Soundness

In order to prove that each derivable judgement is semantically valid, we first define a *relativized* notion of validity.

Definition 8 (*Relativized Validity*). Specification A is *valid for e at depth n*, written $\models_n e : A$, if

$$(m \leq n \ \wedge \ E \vdash h, e \Downarrow_m h', v, p) \ \longrightarrow \ A\ E\ h\ h'\ v\ p.$$

Here, the judgement $E \vdash h, e \Downarrow_m h', v, p$ refers to an operational semantics that extends the semantics given in Section 2.4 by an explicit index which models the derivation height. Note that this counter is only used for metareasoning and is independent from the resources. It is immediate to show the equivalence of the two semantics and we omit the details.

The counter n in Definition 8 restricts the set of pre- and post-states for which A has to be fulfilled. It is easy to show that $\models e : A$ is equivalent to $\forall n.\ \models_n e : A$, and that relativized validity is downward closed, i.e. that for $m \leq n$, $\models_n e : A$ implies $\models_m e : A$.

Like unrestricted validity, relativized validity may be generalized to contexts:

Definition 9 (*Relativized Context Validity*). Context Γ is *valid at depth n*, written $\models_n \Gamma$, if for all $(e, A) \in \Gamma$, $\models_n e : A$ holds. Assertion A is *valid for e in context Γ at depth n*, denoted $\Gamma \models_n e : A$, if $\models_n \Gamma$ implies $\models_n e : A$.

Using the index and relativized validity, proofs may be carried out by induction on the derivation height of the operational semantics; related proofs have appeared elsewhere [25,21,26].

In particular, we can prove the following lemma:

Lemma 10. *For $\models_n \Gamma$ and $\Gamma \cup \{\texttt{call}\, f, A\} \rhd body_f : \Theta(A, f)$, let*

$$\models_m (\Gamma \cup \{\texttt{call}\, f, A\}) \longrightarrow \models_m body_f : \Theta(A, f)$$

hold for all m. Then $\models_n \texttt{call}\, f : A$.

Proof. By induction on n. □

Similar results hold for static and virtual method invocations. From this, the following result may be proved:

Lemma 11. *If $\Gamma \rhd e : A$ then $\forall n.\ \Gamma \models_n e : A$*

Proof. By induction on the derivation of $\Gamma \rhd e : A$. □

Finally, the soundness statement is obtained from Lemma 11 by unfolding the definitions of (relativized) validity.

Theorem 12 (*Soundness*). *If $\Gamma \rhd e : A$ then $\Gamma \models e : A$.*

In particular, an assertion that may be derived using the empty context is valid: $\emptyset \rhd e : A$ implies $\models e : A$.

3.7. Completeness

The soundness of a program logic ensures that derivable judgements assert valid statements with respect to the operational semantics. Soundness is thus paramount to a trustworthy proof-carrying code system. In contrast, completeness of program logics has hitherto been mostly of metatheoretical interest. For the intended use as the basis of MRG's hierarchy of program logics, however, this metatheoretical motivation is complemented by a pragmatic motivation. The intention of encoding (possibly yet unknown) high-level type systems as systems of derived assertions requires that any property that (for a given notion of validity) holds for the operational semantics be indeed provable. Partially, this requirement concerns the expressiveness of the assertion language, which, thanks to our choice of shallow embedding, is guaranteed, as any HOL-definable predicate may occur in assertions. On the other hand, the usage of a logically incomplete ambient logic such as HOL renders the program logic immediately incomplete itself, via the rule of consequence. The by now accepted idea of *relative* completeness [27] proposes to separate reasoning about the program logic from issues regarding the logical language. In particular, the side condition of rule VCONSEQ only needs to *hold* in the meta-logic, instead of being required to be provable.

In our setting, the role of *most general formulae*, originally introduced by Gorelick [28] to prove completeness of recursive programs, is played by *strongest specifications*. These are those assertions that are satisfied exactly for the tuples of the operational semantics.

Definition 13 (*Strongest Specification*). The strongest specification for e is defined by

$$SSpec(e) \equiv \lambda\, E\, h\, h'\, v\, p.\, E \vdash h, e \Downarrow h', v, p.$$

It is immediate that strongest specifications are valid

$$\models e : SSpec(e)$$

and imply all other valid specifications:

$$\text{If } \models e : A \text{ and } SSpec(e)\ E\ h\ h'\ v\ p \text{ then } A\ E\ h\ h'\ v\ p. \tag{8}$$

The context Γ_{strong} associates to each function label and each method declaration in the global program P its strongest specification

$$\Gamma_{strong} \equiv \left\{ (e, SSpec(e)) \mid \begin{array}{r} \exists f.\, e = \mathtt{call}\, f \ \vee\ \exists\, c\, m\, \overline{a}.\, e = c.m(\overline{a}) \\ \vee\ \exists\, x\, m\, \overline{a}.\, e = x \cdot m(\overline{a}) \end{array} \right\}.$$

By induction on e, we prove:

Lemma 14. *For any e, we have* $\Gamma_{strong} \rhd e : SSpec(e)$.

This result can be used to show that Γ_{strong} respects the *strongest specification table*,

$$ST_{strong} \equiv (\lambda\, f.\, SSpec(\mathtt{call}\, f), \lambda\, c\, m\, \overline{a}.\, SSpec(c.m(\overline{a})), \lambda\, x\, m\, \overline{a}.\, SSpec(x \cdot m(\overline{a}))).$$

Lemma 15. *We have* $\Gamma_{strong} \models ST_{strong}$.

The proof of this lemma proceeds by unfolding the definitions, using Lemma 14 in the claims for function calls and method invocations.

Next, we prove that

$$\Gamma_{strong} \models ST \text{ implies } \emptyset \rhd e : SSpec(e) \tag{9}$$

for arbitrary specification table ST, by applying rule VCTXT, where the first premise is discharged by Lemma 14 (i.e. Γ is instantiated to Γ_{strong}) and the second premise is discharged by rule MUTREC.

Combining property (9) and Lemma 15 yields $\emptyset \rhd e : SSpec(e)$, from which

Theorem 16 (*Completeness*). *For any e and A,* $\models e : A$ *implies* $\emptyset \rhd e : A$.

follows by rule VCONSEQ and property (8).

4. Program logic for termination

So far the program logic developed for Grail has been a logic for partial correctness. In this section we will develop a program logic for termination, with a judgement $\rhd_T \{P\}\ e\ \downarrow$, to be read as "expression e terminates under precondition P". In formalizing the concept of a precondition, we adopt the same approach as in the core logic and use a shallow embedding. Thus, preconditions are predicates over environments and heaps in the metalogic and have the type $\mathcal{P} \equiv \mathcal{E} \to \mathcal{H} \to \mathcal{B}$.

With this termination logic the total correctness statement that expression e fulfils assertion A and terminates under precondition P is then the conjunction of statements of these two logics: $\emptyset \rhd e : A\ \wedge\ \rhd_T \{P\} e\ \downarrow$. In contrast to the usual approach of adding the termination-guaranteeing side-conditions directly into the rules for function calls and method invocations,[8] our approach has the advantage of being modular: we do not have to modify the underlying logic for partial correctness at all.

Our goal with this logic is to characterize programs that terminate correctly and therefore we do not distinguish between different sources of non-termination. We note, however, that most cases of program abortion are already checked in earlier stages of code-generation, in particular during bytecode verification. Duplicating these checks in

[8] One notable exception is [29], where a termination logic is built on top of a Hoare-Logic for a while-language.

our logic would only obfuscate the rules and run contrary to our design goal of a modular structure. As ongoing further work we are formalizing a notion of "partial termination" in order to reason about secure behaviour of non-terminating programs and we elaborate on that in Section 7. To add confidence to the results in this section, we have formalized the termination logic and its soundness proof in Isabelle/HOL.

In the rules for let and for composition we will need a form of semantic "implication", relating the pre-condition P of the construct to the precondition P' for e_2, possibly after a value binding to the variable x. We express this as an assertion in the partial correctness logic and define the following combinator.

Definition 17. A *pre-condition implication* of two assertions P, P' with a binding to variable x, written $P \longrightarrow_{\langle x:=\rangle} P'$, is defined as follows:

$$\lambda E\ h\ h'\ v\ p.\ P\ E\ h\ \longrightarrow\ (\exists i.\ v = i\ \wedge\ P'\ E\langle x := i\rangle\ h').$$

Recall that the operation $E\langle x := i\rangle$ performs an update of the variable x by the value i. A similar combinator $\longrightarrow_{\langle \_:=\rangle}$ expresses a precondition implication without value-binding.

4.1. Proof system

We now introduce the proof system for termination which defines the judgement $\rhd_T\{P\}\ e\ \downarrow$. While the core logic uses contexts to collect assumptions on methods and functions, we rely here on metalevel implication for this purpose: to prove that a call f terminates for all inputs one must exhibit a measure function and prove for all n that the body of f terminates for inputs of measure up to n assuming that calls to f terminates for inputs of measure less than n. Unlike the partial case such a rule can be justified by metalevel induction on the termination measure.

Note that a similar use of metalevel implication would be unsound for partial correctness. For each function (or method) f it trivially holds that either f satisfies the partial correctness assertion ϕ or else it is the case that its body satisfies it, assuming that any call to f satisfies ϕ by "ex falso quodlibet". Thus, in the partial correctness case the access to assumptions about calls must be suitably restricted (e.g. using contexts) so as to rule out metalevel case distinctions as above, whereas no such restriction is necessary for termination logic due to the outside quantification over the measure.

Thus, we can use a shallow encoding of object-logic contexts, or, in other terms, a form of hypothetical-parametrical judgements as in type theory, where the meta-logic context encodes the object one. The drawback is that termination does not have an induction principle, but the encoding abstracts from explicit context management operations such as axiomatic lookup or weakening, which are delegated to the metalogic.

4.2. Rules

$$\frac{\forall E\ h.P'\ E\ h\ \longrightarrow\ P\ E\ h \qquad \rhd_T\{P\}\ e\ \downarrow}{\rhd_T\{P'\}\ e\ \downarrow}\ (\textsc{TConseq})$$

$$\frac{}{\rhd_T\{P\}\ \texttt{null}\ \downarrow}\ (\textsc{TNull}) \qquad \frac{}{\rhd_T\{P\}\ \texttt{imm}\ i\ \downarrow}\ (\textsc{TImm})$$

$$\frac{}{\rhd_T\{P\}\ \texttt{var}\ x\ \downarrow}\ (\textsc{TVar}) \qquad \frac{}{\rhd_T\{P\}\ \texttt{prim}\ op\ x\ y\ \downarrow}\ (\textsc{TPrim})$$

$$\frac{\forall E\ h.\ P\ E\ h\ \longrightarrow\ E\langle x\rangle \neq \mathsf{null}}{\rhd_T\{P\}\ x.t\ \downarrow}\ (\textsc{TGetF}) \qquad \frac{\forall E\ h.\ P\ E\ h\ \longrightarrow\ E\langle x\rangle \neq \mathsf{null}}{\rhd_T\{P\}\ x.t{:=}y\ \downarrow}\ (\textsc{TPutF})$$

$$\frac{}{\rhd_T\{P\}\ c \diamond t\ \downarrow}\ (\textsc{TGetS}) \qquad \frac{}{\rhd_T\{P\}\ c \diamond t{:=}y\ \downarrow}\ (\textsc{TPutS})$$

$$\frac{}{\rhd_T\{P\}\ \texttt{new}\ c\ [\overline{t_i := x_i}]\ \downarrow}\ (\textsc{TNew})$$

$$\frac{\rhd_T\{\lambda E\ h.\ P\ E\ h\ \wedge\ E\langle x\rangle = \mathsf{true}\}\ e_1\ \downarrow \quad \rhd_T\{\lambda E\ h.\ P\ E\ h\ \wedge\ E\langle x\rangle = \mathsf{false}\}\ e_2\ \downarrow \quad \forall E\ h.\ P\ E\ h\ \longrightarrow\ E\langle x\rangle \in \{\mathsf{true}, \mathsf{false}\}}{\rhd_T\{P\}\ \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ \downarrow} \quad \text{(TIF)}$$

$$\frac{\rhd_T\{P\}\ e_1\ \downarrow \quad \rhd_T\{P'\}\ e_2\ \downarrow \quad \rhd e_1 : P \longrightarrow_{\langle \_ := \rangle} P'}{\rhd_T\{P\}\ e_1\ ;\ e_2\ \downarrow} \quad \text{(TCOMP)}$$

$$\frac{\rhd_T\{P\}\ e_1\ \downarrow \quad \rhd_T\{P'\}\ e_2\ \downarrow \quad \rhd e_1 : P \longrightarrow_{\langle x := \rangle} P'}{\rhd_T\{P\}\ \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2\ \downarrow} \quad \text{(TLET)}$$

$$\frac{\forall n.\ \rhd_T\{\lambda E\ h.\ \exists m.\ m < n\ \wedge\ P\ m\ E\ h\}\ \mathtt{call}\ f\ \downarrow \longrightarrow\ \rhd_T\{P\ n\}\ body_f\ \downarrow}{\rhd_T\{\lambda E\ h.\ \exists n.\ P\ n\ E\ h\}\ \mathtt{call}\ f\ \downarrow} \quad \text{(TCALL)}$$

$$\frac{\forall n.\ \rhd_T\{\lambda E\ h.\ \exists m.\ m < n\ \wedge\ P\ m\ E\ h\}\ c.m(\overline{a})\ \downarrow \longrightarrow \rhd_T\{\lambda E\ h.\ \exists E'.\ E = Env(\mathtt{self} :: pars_{c,m}, \mathsf{null} :: \overline{a}, E')\ \wedge\ P\ n\ E'\ h\}\ body_{c,m}\ \downarrow}{\rhd_T\{\lambda E\ h.\ \exists n.\ P\ n\ E\ h\}\ c.m(\overline{a})\ \downarrow} \quad \text{(TSINV)}$$

$$\frac{\forall n.\ \rhd_T\{\lambda E\ h.\ \exists m\ l.\ m < n\ \wedge\ E\langle x\rangle = \mathsf{Ref}\ l\ \wedge\ h(l) = c\ \wedge\ P\ m\ E\ h\}\ x \cdot m(\overline{a})\ \downarrow \longrightarrow \rhd_T\{\lambda E\ h.\ \exists E'.\ E = Env(\mathtt{self} :: pars_{c,m}, x :: \overline{a}, E')\ \wedge\ P\ n\ E'\ h\}\ body_{c,m}\ \downarrow}{\rhd_T\{\lambda E\ h.\ \exists n\ l.\ E\langle x\rangle = \mathsf{Ref}\ l\ \wedge\ h(l) = c\ \wedge\ P\ n\ E\ h\}\ x \cdot m(\overline{a})\ \downarrow} \quad \text{(TINV)}$$

The TCONSEQ rule allows us to weaken the precondition P' to P if it is a logical consequence of P'. For the leave cases the only possibility of non-termination is a null-reference in a get- or put-field operation. The operational semantics gets stuck in this case. The conditional rule, TIF, requires termination of both branches under the added knowledge of the value of the boolean header. In the rules for composition, TCOMP and for let, TLET, we use a side condition in the partial correctness logic to establish a link between the overall precondition P and the precondition P' before the second subexpression.

To deal with recursion, the rules TCALL, TSINV and TINV use a well-founded relation $<$ and require that termination of the body must be provable, assuming termination of a Call (or Invoke) for an arbitrary, smaller measure, i.e. for $m < n$. Note that any well-founded ordering can be used in these rules. It is also worth noting that this formalization, with an inner existential quantifier in the precondition, is easier to apply than the version that uses an outer universal quantification over m, since the value binding is deferred until the precondition is exposed in the proof. As usual for such measures, the variable n is existentially quantified in the conclusions of the TCALL, TSINV and TINV rules, since it captures the value of the measure at the particular call point. The rules for static and dynamic method invocation account for the modification of the environment, by applying the *Env* operator to the outer environment E'. The TINV rule contains another clause in its precondition, stating that the variable x points to an instance of class c.

4.3. Soundness

We now define the notion of termination semantically as follows:

Definition 18 (*Termination*). The expression e *terminates under the pre-condition* P, written $\models_T \{P\}\ e\ \downarrow$, iff for all environments E and heaps h

$$P\ E\ h\ \longrightarrow\ \exists h'\ v\ p.\ E \vdash h, e \Downarrow h', v, p.$$

To spell it out, a Grail expression e terminates under a precondition P, if for all states, i.e. environments E and heaps h that fulfil P, a final state comprised of heap h', value v and resources p exists in the operational semantics.

We now prove that the rules for the proof system in Section 4.2 are sound w.r.t. this definition of termination, i.e. we will prove them as lemmas on the semantic definition of termination.

Theorem 19 (*TSoundness*). *The relation* $\models_T \{P\}\ e\ \downarrow$ *is closed under the proof rules for* $\rhd_T\{P\}\ e\ \downarrow$ *in Section* 4.2.

Proof. By cases on e: the leaf cases are straightforward. In the TCOMP and TLET cases, soundness of the partial correctness logic is used to propagate the precondition P through the let header. Likewise for the header in the conditional. The TCALL, TSINV and TINV cases are proven by well-founded induction of the predicate $\lambda n. \forall E\ h. P\ n\ E\ h \longrightarrow (\exists h'\ v\ p.\ E \vdash h,\ e' \Downarrow h',\ v\ ,\ p)$ over n, where e' is $\texttt{call}\ f$, $c.m(\overline{a})$, or $x \cdot m(\overline{a})$, respectively. □

4.4. Completeness

In this section we present a proof of completeness for the termination logic. In the proof we only cover functions, but the extension to methods can use the same techniques for a suitably modified definition of the predicate IH_k used below. Note that we instantiate the well founded relation $<$ to "strictly less" on the naturals numbers. The overall structure of the completeness proof is as follows: as most general formulae we use pairs of expression e and its weakest precondition $wp\ e$. Thanks to the consequence rule it suffices to show $\rhd_T \{wp\ e\}\ e\ \downarrow$. We reduce this to a proof of Lemma 24, which augments the weakest precondition with a descending chain of naturals; this lemma can be proven by induction over the chain.

Definition 20. The *weakest precondition* of an expression e, written $wp\ e$, and the *bounded weakest precondition* of an expression e for $n \in \mathbb{N}$, written $wp'\ e\ n$, are defined as follows:

$$wp\ e \equiv \lambda\ E\ h.\ \exists m.\ \exists\ h'\ v\ p.\ E \vdash h, e \Downarrow_m h', v, p \tag{10}$$

$$wp'\ e\ n \equiv \lambda\ E\ h.\ \exists\ m \leq n.\ \exists\ h'\ v\ p.\ E \vdash h, e \Downarrow_m h', v, p. \tag{11}$$

The index m in the operational semantic judgement refers to the height of the derivation, as in Definition 8. It is easy to show that e terminates under pre-condition $wp\ e$, i.e. $\models_T \{wp\ e\}\ e\ \downarrow$, and that it is the weakest such precondition, i.e. $\models_T \{P\}\ e\ \downarrow \longrightarrow \forall E\ h. P\ E\ h \longrightarrow wp\ e\ E\ h$.

In the proof of completeness we use a variant of the call rule that uses an explicit base case rather than a course-of-values induction principle. This rule follows directly from TCALL and it goes back to Sokołowski [30].

$$\frac{\begin{array}{c}\forall E\ h. \neg P\ 0\ E\ h \\ \forall n.\ \rhd_T\ \{P\ n\}\ \texttt{call}\ f\ \downarrow \longrightarrow \rhd_T\ \{P\ (n+1)\}\ body_f\ \downarrow\end{array}}{\rhd_T \{\lambda E\ h.\ \exists n.\ P\ n\ E\ h\}\ \texttt{call}\ f\ \downarrow} \quad \text{(TCALL')}$$

The completeness theorem establishes the fact that if $\models_T \{P\} e \downarrow$ holds, then this can be proven using the proof rules for $\rhd_T$ alone. This is established by reducing it to a proof of $\rhd_T \{wp\ e\}\ e\ \downarrow$ (see Lemma 25), and this lemma is in turn reduced to a proof of the following predicate IH_k, for all k (see Lemma 24). In defining IH_k we assume, without loss of generality, that the function names are enumerated as $f_1, \ldots, f_M$. For the proof it turns out to be crucial to include the decreasing list of measures n_j for the functions f_j in the preconditions of the function calls and of expression e.

Definition 21. For expression e, $k, M \in \mathbb{N}$, $k \leq M$, $n_1, \ldots, n_k \in \mathbb{N}$, let

$$\begin{array}{rcl} IH_k & \equiv & (\forall\ 1 \leq i \leq k.\ \rhd_T \{wp'\ (\texttt{call}\ f_i)\ n_i\ \wedge\ \forall 1 \leq j < i.\ n_j \geq n_{j+1}\}\ \texttt{call}\ f_i\ \downarrow) \longrightarrow \\ & & \rhd_T \{wp'\ e\ n_k\ \wedge\ \forall 1 \leq j < k.\ n_j \geq n_{j+1}\}\ e\ \downarrow. \end{array}$$

We will now prove, by induction over $M - k$, that IH_k holds for all k. The following lemma will be used to prove the base case, i.e. IH_M, and Lemma 23 will be used to prove the induction step.

Lemma 22. *For all expressions e, $n_1, \ldots, n_M \in \mathbb{N}$,*

$$\begin{array}{l} (\forall\ 1 \leq i \leq M.\ \rhd_T \{wp'\ (\texttt{call}\ f_i)\ n_i\ \wedge\ \forall 1 \leq j < i.\ n_j \geq n_{j+1}\}\ \texttt{call}\ f_i\ \downarrow) \longrightarrow \\ \rhd_T \{wp'\ e\ n_M\ \wedge\ \forall 1 \leq j < M.\ n_j \geq n_{j+1}\}\ e\ \downarrow. \end{array}$$

Proof. By structural induction over e. For all non-call cases, apply the syntax-orientated rule, then for each subexpression TCONSEQ, monotonicity of the bounded weakest precondition, and then the induction hypothesis. The call case follows directly via TCONSEQ, monotonicity of the bounded weakest precondition and monotonicity of the $\geq$ chain. □

The following lemma allows us to add information about the function f_{k+1} into the metacontext. This will be used to discharge the implication in IH_{k+1} in the proof of the induction step (see Lemma 24). Note that this step corresponds to the cut rule on an explicit context, which was used in the proof for completeness of the core partial correctness logic.

Lemma 23. *For all $n_1, \ldots, n_{k+1} \in \mathbb{N}$, if IH_{k+1} then*

$$(\forall\, 1 \leq i \leq k.\ \rhd_T \{wp'\ (\texttt{call}\ f_i)\ n_i\ \wedge\ \forall 1 \leq j < i.\, n_j \geq n_{j+1}\}\ \texttt{call}\ f_i\ \downarrow) \longrightarrow$$
$$\rhd_T \{wp'\ (\texttt{call}\ f_{k+1})\ n_{k+1}\ \wedge\ \forall 1 \leq j < k+1.\, n_j \geq n_{j+1}\}\ \texttt{call}\ f_{k+1}\ \downarrow .$$

Proof. We first use TCONSEQ then TCALL'. This leaves the following to prove:

$$\rhd_T \{wp'\ (\texttt{call}\ f_{k+1})\ (n_{k+1}+1)\ \wedge\ \forall 1 \leq j < k+1.\, n_j \geq n_{j+1}+1\}\ body_{f_{k+1}}\ \downarrow,$$

assuming $\rhd_T \{wp'\ (\texttt{call}\ f_{k+1})\ n_{k+1}\ \wedge\ \forall 1 \leq j < k+1.\, n_j \geq n_{j+1}\}\ \texttt{call}\ f_{k+1}\ \downarrow$. This follows via TCONSEQ, using the equality $wp'\ (body_f)\ n\ E\ h = wp'\ (\texttt{call}\ f)\ (n+1)\ E\ h$ and monotonicity of the $\geq$ chain from $\rhd_T \{wp'\ (body_{f_{k+1}})\ n_{k+1}\ \wedge\ \forall 1 \leq j < k+1.\, n_j \geq n_{j+1}\}\ body_{f_{k+1}}\ \downarrow$, which we know from the induction hypothesis IH_{k+1}. □

Lemma 24. *For all $k \in \mathbb{N}, k \leq M$, IH_k holds.*

Proof. The proof of this lemma is by induction over $M-k$. The base case, with $k = M$, follows from Lemma 22, the induction case follows from Lemma 23 and the induction hypothesis. □

Lemma 25. *For all expressions e, the following holds: $\rhd_T \{wp\ e\}\ e\ \downarrow$.*

Proof. By induction over e. The leaf cases are trivial. The cases TCOMP, TLET and TIF use completeness of the core logic and determinism of evaluation of the operational semantics. The TCALL case uses TCONSEQ with Lemma 24. □

The completeness theorem now follows directly from Lemma 25 and TCONSEQ.

Theorem 26 (*TCompleteness*). *If $\rhd_T$ is any relation that validates the proof rules in Section 4.2, then $\models_T \subseteq \rhd_T$.*

4.5. Example

As example program we use a variant of the even/odd program from Section 3.5.5, which uses only functions and decrements x. We emphasize that we can prove its termination directly via the rules in Section 4.2 without any additional rules for mutual recursion.

We need to specify the precondition under which the functions terminate. Typically this involves characterizing the possible parameter values and class membership for the methods involved. The form of the precondition should fit the structure of the precondition in the call rule to be existentially quantified over the measure. In this case we can use the same parameterized precondition for both functions: $P_{e/o} \equiv \lambda n\ E\ h.\ 0 \leq E\langle x\rangle \wedge n = E\langle x\rangle$.

We need to provide a measure for each function, which assures termination. We can read the measure from the form of the above precondition: $M_{e/o} \equiv \lambda E\ h.\ E\langle x\rangle$. We prove termination of even, i.e.

$$\rhd_T \{\lambda E\ h.\ \exists n.\ P_{e/o}\ n\ E\ h\}\ \texttt{call}\ \mathsf{even}\ \downarrow .$$

In the proof we first apply the TCALL rule, adding information about the precondition for even to the metacontext. Only syntax-directed rules are used to traverse the body of the function. When encountering odd we first use TCONSEQ and then prove termination of odd under the precondition $\lambda E\ h.\exists n'.\ P_{e/o}\ n'\ E\ h\ \wedge\ n' = n-1$, which is implied by the precondition at the call point. We use the constraint $n' = n-1$ to relate the measure for even, namely n, to that of odd, namely n', and then apply the TCALL rule. Following the proof idea in the completeness proof, we could use the weaker constraint $n' \leq n$; however, this would yield a longer proof and thus we pick the exact value for this program. When arriving at the second call to even, we use the first clauses in the metacontext, which are:

$$\rhd_T \{\lambda E\ h.\ \exists m.\ m < n\ \wedge\ 0 \leq E\langle x\rangle\ \wedge\ m = E\langle x\rangle\}\ \texttt{call}\ \mathsf{even}\ \downarrow$$
$$\rhd_T \{\lambda E\ h.\ n-1 < n'\ \wedge\ 0 \leq E\langle x\rangle\ \wedge\ n-1 = E\langle x\rangle\}\ \texttt{call}\ \mathsf{odd}\ \downarrow .$$

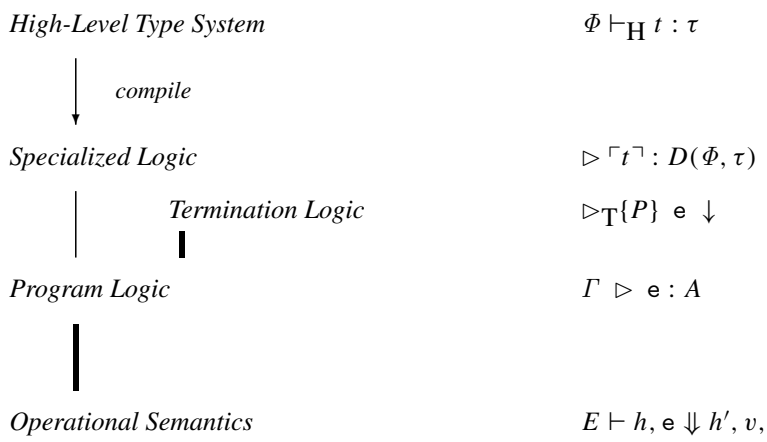

Fig. 6. A family of logics for resource consumption.

These clauses were added by the two invocations of the TCALL rule. We now instantiate the measure to $n-2$ for even. After using TCONSEQ only the side-conditions generated by the syntax-orientated rules remain. These are solved using soundness of $\triangleright$, rules of the operational semantics and some auxiliary lemmas, formalizing non-interference of variables in the evaluation of subexpressions. A case distinction over $n > 0$ is necessary to separate the recursion path from the non-recursive path through the program, and basic arithmetic completes the proof.

5. An infrastructure for resource certification

In this section we describe the infrastructure for certification of resources which we build on top of the program logics presented above. This is based on a multilayered logics approach (shown in Fig. 6), starting from a type system at the top. A more detailed discussion of this infrastructure is given in [31].

While the complexity of proving general program correctness often restricts the research on program verification to only security-critical systems, increasingly complex type systems have found their way into mainstream programming and are accepted as useful tools in software development. Given their complexity, soundness proofs of those type systems can be subtle, and the user has to trust both the proof and the translation of the high-level code into the object-code. One approach to guaranteeing the absence of bad behaviour is to translate types into proofs in a suitably specialized program logic. Going by this route, a formal certificate can be generated and independently checked.

We begin with our (trusted) *annotated operational semantics* that encodes the basic security-sensitive operations (for example, heap allocation if the security policy is bounded heap consumption). The Foundational PCC approach [32] handles proofs directly at this level and thereby reduces the size of the trusted code base (TCB). Likewise, in principle we do not include any of our logics into the TCB, since we formalize the entire hierarchy of logics in a proof assistant together with their soundness proofs. The formal soundness statements can be used as lemmas to establish claims back down to the operational semantics.

Above the operational semantics, we have our general-purpose *program logic* for partial correctness. It serves as a platform around which higher level logics may be unified; this purpose makes logical completeness of the program logic a desirable property whereas it has hitherto been mostly of meta-theoretic interest. Of course, soundness remains mandatory, as the trustworthiness of any application logic defined at higher levels depends upon it. Our soundness and completeness results establish a tight link between operational semantics and program logic, as shown in Fig. 6.

While assertions in the core logic make statements on partial program correctness, as we have seen, the *termination logic* is defined on top of this level to certify termination. This separation improves modularity in developing these logics, and allows us to use judgements of the partial correctness logic when talking about termination.

On top of the general-purpose logic, a *specialized logic* (for example the heap logic of [33]) can be defined to capture the specifics of a particular security policy and its proof mechanism. This logic uses another form of judgement, dubbed *derived assertions*, which reflects the information of the high-level type system. Judgements in the specialized logic have the form $\triangleright \ulcorner \mathtt{t} \urcorner : D(\Phi, \tau)$, where the expression $\ulcorner \mathtt{t} \urcorner$ is the result of compiling a high-level term t down to the low-level language, and the information in the high-level type system is encoded in the derived assertion

$D(\Phi, \tau)$ that depends on the high-level typing context Φ and type τ associated to t. Depending on the property of interest, this level may be further refined into a hierarchy of proof systems, for example, if parts of the soundness argument of the specialized assertions can be achieved by different type systems. Of course, to have confidence in the infrastructure, we want to believe that user-level statements in the high-level type system are faithfully translated into the program logic, and that the compilation is correct. It may be that the expansion of a particular derived assertion is comprehensible to the user, but typically both the encoding and compilation functions will be additional points of trust in our infrastructure.

Unlike the general-purpose logic, this logic is not expected to be logically complete, but instead it should provide support for automated proof search or reconstruction. For the heap logic mentioned, this is achieved by formulating a system of derived assertions with a similar level of granularity to the type system itself. However, the rules are expressed in terms of code fragments in the low-level language. Since the side conditions of the typing rules are computationally easy to validate, automated proof search is supported by the syntax-directness of the typing rules. At points where syntax-directness fails – such as recursive program structures – the necessary invariants are provided by translation from the type system.

Finally, on the top level we find the *high-level type system* itself, that encodes information on resource consumption in a way that can be meaningful to the high-level programmer. In the judgement $\Phi \vdash_H t : \tau$, the term t has an (extended) type τ in a context Φ. This level is of closer relevance to programming languages research, and numerous type-based inferences have been suggested for establishing safety and resource usage policies. The system we worked with in the heap logic of [33] was Hofmann & Jost's type system for heap usage [34], but our infrastructure and methodology is quite general.

5.1. Example

For illustration purposes, let us instantiate this hierarchy with a a security property of "well-formed datatypes", related to memory safety polices typically considered in PCC. To model datatypes, we use a version of the previously defined predicate $h \models_\tau a$, expressing that an address a in heap h is the start of a (high-level) datatype τ. In the core program logic we can express the fact that a method $c.m_f$ compiled from the high-level function $f :: \tau\ list \to \tau\ list$ preserves (the representation of) a well-formed data-type as follows:

$$\triangleright\, c.m_f(x) \ :\ \lambda E\ h\ h'\ v.\ h \models_{list} E\langle x\rangle \ \longrightarrow\ h' \models_{list} v.$$

To abstract over the details of modelling data-structures, we construct a specialized logic for this security property, by restricting the form of the assertions to reflect the high-level property to be formalized. These derived assertions $D(\Phi, \tau)$ must have the following property for a program term t and a high-level type τ:

$$\Phi \vdash_H t : \tau \implies \triangleright\ulcorner t \urcorner : D(\Phi, \tau).$$

The definition of the derived assertion carries with it the high-level types of the variables, and accesses low-level predicates, which are needed to express the security policy but should be hidden in the specialized logic itself:

$$\begin{aligned} D(\{x : \tau\ list, y : \tau\ list\}, \tau\ list) \quad \equiv \quad \lambda E\ h\ h'\ v\ p. \quad & h \models_{list} E\langle x\rangle \ \wedge\ h \models_{list} E\langle y\rangle \ \longrightarrow \\ & h' \models_{list} E\langle x\rangle \ \wedge\ h' \models_{list} E\langle y\rangle \ \wedge\ h' \models_{list} v. \end{aligned}$$

Based on this format we could phrase a rule for a high-level list-cons operator, which abstracts from the representation predicates:

$$\frac{\triangleright\ulcorner t_1 \urcorner : D(\Phi, \tau) \qquad \triangleright\ulcorner t_2 \urcorner : D(\Phi, \tau\ list)}{\triangleright\ulcorner cons(t_1, t_2) \urcorner : D(\Phi, \tau\ list)}$$

The soundness of the rules in the specialized logic can then be proved once and for all on top of the core logic. Each rule roughly corresponds to a case in the type soundness proof at the high level. The main advantages of the specialized logic are proofs in this logic being largely syntax-directed with simpler side conditions. It is in fact possible to turn (high-level) type checking into an (effective) tactic in a proof assistant. In comparison, we found that using an interactive prover to prove such assertions directly becomes very complicated for non-trivial programs. Even with a verification condition generator, side conditions arise that are difficult to handle manually and too challenging for current automatic theorem provers.

6. Related work

There has been an outstanding amount of research on formalizing the safety of Java, the Java Virtual Machine and the JavaCard environment; see [35] for a review up to 2001. For our purposes, we review only work related to program logics, which we can divide in roughly three categories: imperative and object-orientated logics, pointer logics and bytecode logics. Note, however, that none of this related work contains any formalized account of resources.

6.1. Imperative and object-orientated logics

Most closely related to our work on the metatheoretical side are Nipkow's implementation of Hoare logic [25], the Java-light logic by von Oheimb [36], Kleymann's thesis [21], and Hofmann's [26] work on completeness of program logics. The formalized logic by Nipkow in [25] concerns a while language with parameterless functions, with proofs of soundness and completeness. Although several techniques we use in our proofs are motivated by this work, we have made advances on the treatment of mutual recursion and adaptation. In particular, in covering mutual recursion we do not specify a separate derivation system but only use a derived rule MUTREC; this make the logic smaller and easier to handle when proving soundness and completeness.

Several options for formalizing either VDM or Hoare-style program logics were explored by Kleymann in his thesis [21]. In particular, his work demonstrates how to formalize an adaptation rule that permits to modify auxiliary variables. Kleymann's logic for total correctness of a while language with functions is closely related to our termination logic. In particular, the call rule is similar, but he does not cover methods. We used these results, together with our own previous experience on VDM-style logics [37] to design the Grail program logic. The techniques used in our completeness proof are based on those by one of the authors in [26].

The program logic for Java-light by von Oheimb [36] is encoded in Isabelle and proven sound and complete. It covers more object-orientated features, but it works on a higher level than our logic for a bytecode language and does not address resources. Moreover, because of the focus on metatheoretical properties, it is hardly suitable for concrete program verification, as it can also be seen from the only example provided. In the same paper von Oheimb discusses an alternative calculus for total correctness with a rule for method invocation that allows a similar handling of mutual recursion as our system. Both Kleymann and von Oheimb develop logics combining functional correctness and termination in one logic, yielding a far more complicated system than our termination logic.

Elsewhere, de Boer et al. [38,39] present a sound and complete Hoare-style logic for a sequential object-orientated language with inheritance and subtyping. In contrast to our approach, the proof system employs a specific assertion language for object structures, a WP calculus for assignment and object creation, and Hoare rules for method invocation. The approach is heavily based on syntactical substitutions and the logic is not compositional. Recently a tool supporting the verification of annotated programs (flowcharts) yielding verification conditions to be solved in HOL has been produced [40]. This also extends to multi-threaded Java [41], where also a non-formalized proof of the soundness and completeness of its assertion-based proof system is provided.

Abadi and Leino combine a program logic for a simple object-orientated language with a type system [42,43]. The language supports subclassing and recursive object types. In a judgement, specifications as well as types are attached to expressions. It uses a global store model, with the possibility of storing pointers to arbitrary methods in objects. This logic is more expressive than our program logic, but it is incomplete.

Homeier [29] also develops a termination logic separately from his partial correctness Hoare-logic for a simple while-language with procedures. His treatment of mutual recursion builds on procedure entrance specifications that relate the state at the beginning of procedure with that at a call point in the procedure. These entrance specifications can be used to encode a descending chain, as we use it in the completeness proof, but in terms of logics his work uses heavier machinery (a separate logic for entrance specifications). In contrast to our work, he doesn't give formal completeness results for his termination logic.

Several related projects aim at developing program logics for subsets of Java, mainly as tools for program verification. Müller and Poetzsch-Heffter present a sound Hoare-style logic for a Java subset [22]. Their language covers class and interface types with subtyping and inheritance, as well as dynamic and static binding, and aliasing via object references, which has been implemented in the *Jive* tool [44]. As part of the *LOOP* [45] project, Huisman and Jacobs [46] present an extension of a Hoare logic that includes means for reasoning about abrupt termination and side-effects, encoded in PVS and Isabelle. In [47] a set of sound rules for the sequential imperative fragment of Java

are given, based on the Java Modelling Language (JML). This combines all possible termination modes in a single logic.

Krakatoa [48] is a tool for verifying JML-annotated Java programs that acts as front-end to the *Why* system [49], using Coq to model the semantics and conduct the proofs. *Why* produces proof obligations for programs in imperative-functional style via an interpretation in a type theory of effects and monads. JACK [50] is based on a weakest preconditions calculus, which is asserted rather than semantically derived from a machine model. It generates proof obligations from annotated Java sources. The proof obligations can be discharged using different systems, from automated tools such as *Simplify* to interactive theorem provers. Much effort is invested in making the system usable by Java programmers by means of an Eclipse plug-in and a proof-obligation viewer. Recently this has been extended to a logic for sequential bytecode, where annotations are expressed in a Bytecode Modeling Language, which is the target of a JVM compiler [51]. The logic is based on a weakest precondition calculus, whose soundness is reported to have been proven w.r.t. to the operational semantics of the JVM in [52]. In addition to defining a predicate transformer for each bytecode, the approach requires the computation of the WP of the whole program using its control flow graph.

In the context of the *KeY* project [53], Beckert [54] presents a sequent calculus for a version of Dynamic Logic dedicated to the JavaCard language with judgments of the form $\Gamma \vdash \langle \mathbf{e} \rangle \phi^U$, where e is a program, Γ encodes preconditions and U is a state update used to cope with aliasing. Loops are dealt with by unrolling and method calls by symbolic execution of the body. Soundness and completeness are not proven. This has been integrated with the *KeY* interactive prover and applied to security properties by Mostowski [55] (for example JavaCard runtime exception thrown at top level). An attempt is made to modularize the approach to methods by utilizing pre and post-condition rather than execution of the body. This approach was also extended to deal with JavaCard transactions by introducing a *trace* modality [56].

Also following a dynamic logic approach, the *KIV* system [57] has been used to formalize operational semantics and a sound dynamic logic for full JavaCard, based on strongest postconditions [58]. This work treats a language where expressions and blocks are flattened (a sort of ANF), new instructions are added, for example, to mark the end of a block; judgments in the logic have the form $\Gamma \vdash \langle H; e \rangle \phi, \Delta$, where H is the initial heap. Program rules may be applicable also on the left hand side of the turnstile, yielding a rich calculus with over 50 rules, best suited to interactive verification. The encoding is not definitional, but relies on many axioms over operations such as replacement of variables, flattening of blocks etc.

Intentionally less expressive than program logics, there are several systems orientated towards *lightweight* static validation. The main current one is ESC/Java2 [59], a tool that attempts to find common runtime errors (such as null references) in JML-annotated Java programs by static analysis of the program code and its formal annotations. It trades off soundness (and completeness) in favour of full automation.

6.2. Pointer logics

A related direction is the verification of pointer programs. Prominent recent work here builds on the ideas of Separation Logic [60]. We chose not to follow this directly, although our specifications certainly include the encoding of separation principles. We can speculate that the adoption of Separation Logic's frame rule may simplify our specifications in the core logic, as well as the interpretation of the derived assertions. However, given the well-known issues with completeness and complexity, current efforts in the automation of Separation Logic are geared towards decision procedures for sub-languages [61]. Our approach instead considers a complete low-level logic, but gets a handle on automation by considering restricted higher-level encoded logics.

Before Separation Logic, some of the first formal verification of pointer programs in [62] (and later [63]) used a model where the store is incorporated in the assertion logic. More recent is the verification of several algorithms, including list manipulating programs and the Schorr-Waite graph-marking algorithm, by Bornat [64] using the Jape system. This approach employs a Hoare logic for a while-language with components that are semantically modelled as pointer-indexed arrays. Separation conditions are expressed as predicates on (object) pointers. [65] uses a Hoare logic in the style of Gordon [66] to reason about pointer programs in a simple while-language, including a declarative proof of the correctness of the Schorr-Waite algorithm. An Isabelle/HOL implementation of separation logic following the previous paper is presented in [67], although the author reports proofs (typically in-place reversal) to be slightly more complicated than in [65].

6.3. Bytecode logics

Most previous work on bytecode logics work directly on ordinary bytecode (albeit with certain restrictions), rather than abstracted forms such as our Grail language. Logics are typically based on weakest precondition calculi. Only two [68,69] have a formalized verification. Bannwart and Müller [70] present a sound and complete bytecode logic for a fragment of sequential JVML — 15 instructions, although under some "well-formedness" restrictions, namely only virtual methods with one parameter are allowed, always yielding a value and ending with a `return`, which cannot occur anywhere else. The scenario is proof-transforming compilers, i.e. compilers that modify source-level correctness proofs. Thus, the system shares the object model and many of the structural rules with [22]. The unstructured nature of bytecode is dealt with by the notion of *instruction specification* and its WP calculus, close to the proposals of Benton [71]. Quigley [68] is mainly concerned with reconstructing high-level control structures such as loops in the bytecode: hence her Hoare rules are somewhat complex, limited (no method calls) and not appropriate for a PCC scenario.

In [69], the authors instantiate the PCC framework of [72] to Jinja bytecode (a dozen instructions including exception handling) and a safety policy concerning arithmetic overflow. Provability is defined semantically and a generic sound and complete verification condition generator based on flow-graph analysis produces annotations in a dedicated assertion language, basically first-order arithmetic with stack primitives. As the framework is generic, other safety policies are possible, provided an appropriate language satisfying certain restrictions is given. The user is required to provide the invariants, although [73] presents preliminary results to automatically extract annotations from untrusted static analysers (in this case, interval analysis).

Finally, Moore has used ACL2 to verify the correctness of simple programs in a fragment of the JVM, directly from the operational semantics, by encoding an interpreter for the byte code as a LISP function [74]. In [75] the same approach was used to prove some metatheoretical properties of the JVM.

7. Conclusion

In this article we have presented program logics for a compact representation of the Java Virtual Machine Language, based on a formalization in Isabelle/HOL. Founded on an operational semantics that includes a flexible notion of resources, we first defined a sound and complete logic of partial correctness, in which pre- and post-conditions are combined to yield specifications that are relations over the components of the operational semantics. We presented derived rules for mutually recursively program fragments and method invocation with parameter adaptation, whose proofs are based on admissible cut rules. The usefulness of these rules was demonstrated on non-trivial examples, including the verification of quantitative specifications and other non-functional properties.

In the second part of this article, we exhibited a logic for termination which relies on partial correctness assertions formulated in the core logic as side conditions of certain rules. Hence we achieve a modular treatment of termination, which leaves the underlying partial logic unchanged. Noteworthy technical features of the termination logic are twofold. First, we use a meta-logical management of context stemming from jumps and method invocations, thusly simplifying the judgement and the proofs. Second, we again have a simplified treatment of mutual recursion, without needing a special proof system or rule. The completeness proof for the termination logic formally justifies that no special treatment of mutual recursion is required here either.

7.1. Discussion and future work

Thanks to the formalized proofs of soundness, we know that both of our program logics are trustworthy bases for a proof-carrying code system such as the one we developed in the *Mobile Resource Guarantees* project [2]. Our approach of implementing a hierarchy of logics addresses a critical issue in the design of PCC systems, namely the trade-off between expressiveness and automation. Indeed, the proofs of the sample programs described in this article appear difficult to automate, and contain typically hundreds of proof steps. To exploit the structure available in a typed high-level language, we use a *derived* proof system that amounts to an interpretation of a high-level type system in the logic, with respect to a compilation strategy. We demonstrated this technique in [33], which gave an interpretation of Hofmann–Jost's LFD system [34]. The soundness proof of a derived logic can hide the complexity of a low-level direct proof. For example, instantiations of (existential) quantifiers and unfolding of datatype representation

predicates are performed only once during the derivation of the derived rules. The application of the derived rules is almost entirely syntax-directed involving side conditions that are generally much easier to discharge. Complementing the communication of typing derivations, the role of type systems in the process of certificate generation becomes that of deriving either invariants (in our formalization, method specifications) or other hints that suffice for the prover to reconstruct proofs. Current and future work in this direction will aim to represent type systems for a variety of resources, and their modular combination.

The recently started MOBIUS (Mobility, Ubiquity and Security) project, an IST FET Integrated Project[9] aims to develop a PCC infrastructure for properties relating to trust and security in a global computing context. The program logic component of MOBIUS builds partly on work described here. In particular, MOBIUS plans a similar strategy of using layers of logics to derive certificates from static analyses and high-level annotations, and ideally, formalized soundness proofs based on a core logic. MOBIUS aims to cover the whole JVM and treat concurrency. This requires modelling of an explicit operand stack and adding invariants which are required to hold throughout the execution of method, whether terminating or not. Like our termination logic, this requires explicit preconditions.

We are generalizing work on the termination logic to formalize other security properties with a novel approach, which we call "partial termination" [76]. The rationale is that classical termination might be a too strong property for the purpose of guaranteeing resource-bounded computation. It prohibits, for example, the treatment of demon processes that are designed to run forever, but should not consume resources. Furthermore, the necessity of defining a measure in the termination logic complicates automatic proofs of termination. The partial termination approach proceeds in two stages. First, insecure behaviour is axiomatized. Then a logic for security is defined on top of it. For example, if a function calls should be restricted to a certain class of functions, possibly under side-conditions of parameter size, we add an axiom for this case. Other rules in the logic are oblivious to the underlying security policy and only propagate the information through the program. We envisage that this will be a superior approach, since it abstracts away from explicit measures.

Regarding resource algebras, we want to explore further ways of structuring the operations and exploiting them, including formal relations to static notions such as effects; this is started in [13]. There are intriguing connections with work elsewhere, especially concerning the traces collected by resource algebras such as $\mathcal{R}^{\mathsf{InvTr}}$ and $\mathcal{R}^{\mathsf{HpTr}}$. For example, Schneider's *EM policies* [18] (security policies that are enforceable by monitoring system execution) and the associated specification formalism, *security automata*, appear well-suited to be formulated as (variants of) $\mathcal{R}^{\mathsf{InvTr}}$ or other resource algebras, which would yield a program logic in which satisfaction of EM policies can be certified. Similarly, Skalka's *history effects* [77] represent a static way to approximate properties of *event histories*, i.e. of abstractions of traces with respect to arbitrary observable properties. Interpreting history effects as labelled transition systems allows Skalka to employ model checking techniques to (statically) verify assertions modelled as formulae in a temporal logic.

As another thread of inquiry, Beckert and Mostowski's *throughout* modality [78] requires the quantified property to be satisfied in all intermediate states of a computation. The satisfaction of this modality appears to follow a similar regime as the validation of the policy that enforces limits of parameter values, as a boolean variable may be used to signal violation. It thus seems possible that we can capture *strong (object) invariants*, that is, properties about all intermediate states of a program such as those described in [56,79].

Finally, we have concentrated in this paper on the expressivity of our logics, so we know they are powerful enough for arbitrary resource specifications. But, as was touched upon in Section 5, we want to be able to write resource usage *policies* in a form that is understandable to the receivers of mobile code, independently of mechanisms such as high-level type systems used for enforcing those policies. We have made some early investigations in this direction recently [80].

Acknowledgments

This research was supported by the MRG project (IST-2001-33149) which was funded by the EC under the FET proactive initiative on Global Computing. We would like to thank all of the researchers who contributed to MRG, including Don Sannella, Ian Stark, Stephen Gilmore, Kenneth MacKenzie, Olha Shkaravska, Matthew Prowse, Michal

[9] See http://mobius.inria.fr/.

Konečný, Robert Atkey and Brian Campbell. We also benefited from discussions with Peter Lee, Tobias Nipkow, and David von Oheimb. Jaroslav Ševčík commented on a draft of the paper.

References

[1] G. Necula, Proof-carrying Code, in: POPL'97 — Symposium on Principles of Programming Languages, Paris, France, 15–17 January 1997, ACM Press, 1997, pp. 106–116.
[2] D. Sannella, M. Hofmann, Mobile resource guarantees, EU OpenFET Project. http://www.groups.inf.ed.ac.uk/mrg/, 2002.
[3] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, I. Stark, Mobile resource guarantees for smart devices, in: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004, in: LNCS, vol. 3362, Springer, 2005, pp. 1–26.
[4] K. MacKenzie, N. Wolverson, Camelot and Grail: Resource-aware functional programming on the JVM, in: Trends in Functional Programing, vol. 4, Intellect, 2004, pp. 29–46.
[5] L. Beringer, K. MacKenzie, I. Stark, Grail: A functional form for imperative mobile code, in: Foundations of Global Computing: Proceedings of the 2nd EATCS Workshop, in: Electronic Notes in Theoretical Computer Science, vol. 85.1, Elsevier, 2003, pp. 1–21.
[6] C. Flanagan, A. Sabry, B.F. Duba, M. Felleisen, The essence of compiling with continuations, in: Proceedings ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993, vol. 28(6), ACM Press, New York, 1993, pp. 237–247.
[7] C. League, V. Trifonov, Z. Shao, Functional Java bytecode, in: Proc. 5th World Conf. on Systemics, Cybernetics, and Informatics, 2001, pp. 1–6. Workshop on Intermediate Representation Engineering for the Java Virtual Machine.
[8] A.W. Appel, Compiling with Continuations, Cambridge University Press, 1992.
[9] X. Leroy, Bytecode verification for Java smart cards, Software Practice and Experience 32 (4) (2002) 319–340.
[10] K. MacKenzie, N. Wolverson, Camelot and Grail: Compiling a resource-aware functional language for the Java virtual machine, in: TFP'03, Symp on Trends in Functional Languages, Edinburgh, 11–12 September 2003.
[11] M. Hofmann, A type system for bounded space and functional in-place update, Nordic Journal of Computing 7 (4) (2000) 258–289.
[12] D.-J. Pym, P.-W. O'Hearn, H. Yang, Possible worlds and resources: the semantics of BI, Theor. Comput. Sci. 315 (1) (2004) 257–305.
[13] D. Aspinall, L. Beringer, A. Momigliano, Optimisation validation, Electron. Notes Theor. Comput. Sci. 176 (3) (2007) 37–59.
[14] A. Chander, J. Mitchell, I. Shin, Mobile code security by Java bytecode instrumentation, in: DARPA Information Survivability Conference & Exposition, DISCEX II, 2001, pp. 1–15.
[15] A. Chander, D. Espinosa, N. Islam, P. Lee, G.C. Necula, Enforcing resource bounds via static verification of dynamic checks, in: Sagiv [81], pp. 311–325.
[16] G. Czajkowski, T. von Eicken, JRes: A resource accounting interface for Java, ACM SIGPLAN Notices 33 (10) (1998) 21–35.
[17] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, A. Momigliano, A program logic for resource verification, in: Proceedings of 17th International Conference on Theorem Proving in Higher Order Logics, TPHOLs2004, in: LNCS, vol. 3223, Springer, Heidelberg, 2004, pp. 34–49.
[18] F.B. Schneider, Enforceable security policies, ACM Transactions on Information and System Security 3 (2000) 30–50.
[19] C.A.R. Hoare, An axiomatic basis for computer programming, Communications of the ACM 12 (10) (1969) 576–580.
[20] C. Jones, Systematic Software Development Using VDM, Prentice Hall, 1990.
[21] T. Kleymann, Hoare logic and VDM: Machine-checked soundness and completeness proofs, Ph.D. Thesis, LFCS, University of Edinburgh, 1999.
[22] A. Poetzsch-Heffter, P. Müller, A programming logic for sequential Java, in: S.D. Swierstra (Ed.), ESOP'99 — European Symposium on Programming, in: LNCS, vol. 1576, Springer, 1999, pp. 162–176.
[23] T. Nipkow, Hoare logics in Isabelle/HOL, in: H. Schwichtenberg, R. Steinbrüggen (Eds.), Proof and System-Reliability, Kluwer, 2002, pp. 341–367.
[24] C. Pierik, F.S. de Boer, Modularity and the rule of adaptation, in: Rattray et al. [82], pp. 394–408.
[25] T. Nipkow, Hoare logics for recursive procedures and unbounded nondeterminism, in: J.C. Bradfield (Ed.), Computer Science Logic, 16th International Workshop, CSL 2002, 11th Annual Conference of the EACSL, Proceedings, in: LNCS, vol. 2471, Springer, 2002, pp. 103–119.
[26] M. Hofmann, Semantik und Verifikation, in: Lecture Notes, TU Darmstadt, 1998.
[27] S.A. Cook, Soundness and completeness of an axiom system for program verification, SIAM Journal on Computing 7 (1) (1978) 70–90. See corrigendum in SIAM Journal on Computing 10, 612.
[28] G.A. Gorelick, A complete axiomatic system for proving assertions about recursive and non-recursive programs, Tech. Rep. 75, University of Toronto, 1975.
[29] P. Homeier, Trustworthy tools for trustworthy programs: A mechanically verified verification condition generator for the total correctness of procedures, Ph.D. Thesis, University of California, 1995.
[30] S. Sokołowski, Total correctness for procedures, in: J. Gruska (Ed.), Mathematical Foundations of Computer Science, in: LNCS, vol. 53, Springer, 1977, pp. 475–483.
[31] M. Hofmann, H.-W. Loidl, L. Beringer, Certification of quantitative properties of programs, in: Lecture Notes of the Summer School on Logical Aspects of Secure Computer Systems, IOS Press, Marktoberdorf, Germany, 2–14 August 2005.
[32] A.W. Appel, Foundational proof-carrying code, in: LICS'01 — 16th Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society, 2001, pp. 247–258.

[33] L. Beringer, M. Hofmann, A. Momigliano, O. Shkaravska, Automatic certification of heap consumption, in: A.V. Franz Baader (Ed.), Logic for Programming, Artificial Intelligence, and Reasoning: 11th International Conference, LPAR 2004, Montevideo, Uruguay, 14–18 March 2005. Proceedings, in: LNCS, vol. 3452, Springer, 2005, pp. 347–362.
[34] M. Hofmann, S. Jost, Static prediction of heap space usage for first-order functional programs, in: POPL'03 — Symposium on Principles of Programming Languages, ACM Press, New Orleans, LA, USA, 2003, pp. 185–197.
[35] P.H. Hartel, L. Moreau, Formalising the safety of Java, the Java virtual machine and Java card, ACM Computing Surveys 33 (4) (2001) 517–558.
[36] D. von Oheimb, Hoare logic for Java in Isabelle/HOL, Concurrency and Computation: Practice and Experience 13 (13) (2001) 1173–1214.
[37] F. Tang, M. Hofmann, Generating verification conditions for Abadi and Leino's logic of objects, in: FOOL9 — Ninth International Workshop on Foundations of Object-Oriented Languages, Portland, OR, 2002, pp. 1–11.
[38] F. de Boer, A WP-calculus for OO, in: Foundations of Software Science and Computation Structures, in: LNCS, vol. 1578, Springer, 1999, pp. 135–149.
[39] C. Pierik, F.S. de Boer, A syntax-directed Hoare logic for object-oriented programming concepts, in: E. Najm, U. Nestmann, P. Stevens (Eds.), FMOODS, in: LNCS, vol. 2884, Springer, 2003, pp. 64–78.
[40] F. d. Boer, C. Pierik, Computer-aided specification and verification of annotated object-oriented programs, in: B. Jacobs, A. Rensink (Eds.), Proceedings of the IFIP TC6/WG6.1 Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems, FMOODS 2002, in: IFIP Conference Proceedings, vol. 209, Kluwer, 2002, pp. 163–177.
[41] E. Ábrahám, F.S. de Boer, W.P. de Roever, M. Steffen, An assertion-based proof system for multithreaded Java, Theoretical Computer Science 331 (2–3) (2005) 251–290.
[42] M. Abadi, R. Leino, A logic of object-oriented programs, in: TAPSOFT '97: Theory and Practice of Software Development, in: LNCS, vol. 1214, Springer, 1997, pp. 682–696.
[43] R. Leino, Recursive object types in a logic of object-oriented programs, Nordic Journal of Computing 5 (4) (1998) 330–360.
[44] J. Meyer, P. Müller, A. Poetzsch-Heffter, Programming and interface specification language of JIVE - specification and design rationale, available from citeseer.ist.psu.edu/86897.html, 2000.
[45] J. van den Berg, B. Jacobs, The LOOP compiler for Java and JML, in: Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001, in: LNCS, Springer, 2001, pp. 299–315.
[46] M. Huisman, B. Jacobs, Java program verfication via a Hoare logic with abrupt termination, in: FASE'00 — Fundamental Approaches to Software Engineering, in: LNCS, vol. 1783, 2000, pp. 284–303.
[47] B. Jacobs, E. Poll, A logic for the Java modeling language JML, in: FASE'01: Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering, Springer, London, UK, 2001, pp. 284–299.
[48] C. Marché, C. Paulin-Mohring, X. Urbain, The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML, Journal of Logic and Algebraic Programming 58 (1–2) (2004) 89–106.
[49] J.-C. Filliâtre, C. Marché, The Why/Krakatoa/Caduceus platform for deductive program verification, in: W. Damm, H. Hermanns (Eds.), CAV, in: Lecture Notes in Computer Science, vol. 4590, Springer, 2007, pp. 173–177.
[50] L. Burdy, A. Requet, J.-L. Lanet, Java applet correctness: A developer-oriented approach, in: K. Araki, S. Gnesi, D. Mandrioli (Eds.), FME 2003: Formal Methods: International Symposium of Formal Methods Europe, in: LNCS, vol. 2805, Springer, 2003, pp. 422–439.
[51] L. Burdy, M. Huisman, M. Pavlova, Preliminary design of BML: A behavioral interface specification language for Java bytecode, in: M.B. Dwyer, A. Lopes (Eds.), FASE, in: Lecture Notes in Computer Science, vol. 4422, Springer, 2007, pp. 215–229.
[52] G. Barthe, M. Pavlova, G. Schneider, Precise analysis of memory consumption using program logics, in: SEFM, IEEE Computer Society, 2005, pp. 86–95.
[53] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, P.H. Schmitt, The KeY tool, Software and Systems Modeling 4 (1) (2005) 32–54.
[54] B. Beckert, A dynamic logic for the formal verification of Java Card programs, in: JavaCard '00: Revised Papers from the First International Workshop on Java on Smart Cards: Programming and Security, Springer, London, UK, 2001, pp. 6–24.
[55] W. Mostowski, Formalisation and verification of Java Card security properties in dynamic logic, in: M. Cerioli (Ed.), Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference 2005, Edinburgh, Scotland, in: LNCS, vol. 3442, Springer, 2005, pp. 357–371.
[56] R. Hähnle, W. Mostowski, Verification of safety properties in the presence of transactions, in: G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, T. Muntean (Eds.), Proceedings, Construction and Analysis of Safe, Secure and Interoperable Smart Devices, CASSIS'04, Workshop, in: LNCS, vol. 3362, Springer, 2005, pp. 151–171.
[57] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, A. Thums, Formal system development with KIV, in: FASE '00: Proceedings of the Third Internationsl Conference on Fundamental Approaches to Software Engineering, Springer-Verlag, London, UK, 2000, pp. 363–366.
[58] K. Stenzel, A formally verified calculus for full Java Card, in: Rattray et al. [82], pp. 491–505.
[59] D.R. Cok, J.R. Kiniry, Esc/Java2: Uniting ESC/Java and JML. Progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system, in: G. Barthe, L. Burdy, M. Huisman, et al. (Eds.), Construction and Analysis of Safe, Secure, and interoperable Smart Devices: Proceedings of the International Workshop, CASSIS 2004, in: LNCS, vol. 3362, Springer, 2005, pp. 108–129.
[60] J.C. Reynolds, Separation logic: A logic for shared mutable data structures, in: LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society, Washington, DC, USA, 2002, pp. 55–74.
[61] J. Berdine, C. Calcagno, P.W. O'Hearn, Symbolic execution with separation logic, in: K. Yi (Ed.), APLAS 2005, in: LNCS, vol. 3780, 2005, pp. 52–68.

[62] D.C. Luckham, N. Suzuki, Verification of array, record, and pointer operations in Pascal, ACM Transactions on Programming Languages and Systems 1 (2) (1979) 226–244.
[63] K.R.M. Leino, Toward reliable modular programs, Ph.D. Thesis, California Institute of Technology, Available as Technical Report Caltech-CS-TR-95-03, 1995.
[64] R. Bornat, Proving pointer programs in Hoare logic, in: MPC '00: Proceedings of the 5th International Conference on Mathematics of Program Construction, Springer, London, UK, 2000, pp. 102–126.
[65] F. Mehta, T. Nipkow, Proving pointer programs in higher-order logic, in: F. Baader (Ed.), Automated Deduction — CADE-19, in: LNCS, vol. 2741, Springer, 2003, pp. 121–135.
[66] M.J.C. Gordon, Mechanizing programming logics in higher-order logic, in: G.M. Birtwistle, P.A. Subrahmanyam (Eds.), Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification), Springer, Banff, Canada, 1988, pp. 387–439.
[67] T. Weber, Towards mechanized program verification with separation logic, in: J. Marcinkowski, A. Tarlecki (Eds.), Computer Science Logic — 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 2004, Proceedings, in: LNCS, vol. 3210, Springer, 2004, pp. 250–264.
[68] C.L. Quigley, A programming logic for Java bytecode programs, in: D.A. Basin, B. Wolff (Eds.), TPHOLs, in: LNCS, vol. 2758, Springer, 2003, pp. 41–54.
[69] M. Wildmoser, T. Nipkow, Asserting bytecode safety, in: Sagiv [81], pp. 326–341.
[70] F. Bannwart, P. Müller, A program logic for bytecode, Electron. Notes Theor. Comput. Sci. 141 (1) (2005) 255–273.
[71] N. Benton, A typed logic for stacks and jumps, Microsoft Research (2004).
[72] M. Wildmoser, T. Nipkow, Certifying machine code safety: Shallow versus deep embedding, in: K. Slind, A. Bunker, G. Gopalakrishnan (Eds.), Theorem Proving in Higher Order Logics, TPHOLs 2004, in: LNCS, vol. 3223, Springer, 2004, pp. 305–320.
[73] M. Wildmoser, A. Chaieb, T. Nipkow, Bytecode analysis for proof carrying code, Electron. Notes Theor. Comput. Sci. 141 (1) (2005) 19–34.
[74] J.S. Moore, Inductive assertions and operational semantics, in: D. Geist, E. Tronci (Eds.), Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, 21–24 October 2003, Proceedings, in: LNCS, vol. 2860, Springer, 2003, pp. 289–303.
[75] J.S. Moore, Proving theorems about Java and the JVM with ACL2, NATO Science Series Sub Series III Computer and Systems Sciences 191 (2003) 227–290.
[76] M. Hofmann, H.-W. Loidl, From partial correctness to total correctness, MRG Deliverable D6g, Institut für Informatik, Ludwig-Maximilians Universität, München, March 2005.
[77] C. Skalka, S.F. Smith, History effects and verification, in: W.-N. Chin (Ed.), APLAS, in: LNCS, vol. 3302, Springer, 2004, pp. 107–128.
[78] B. Beckert, W. Mostowski, A program logic for handling Java Card's transaction mechanism, in: M. Pezzè (Ed.), Proceedings, Fundamental Approaches to Software Engineering, FASE, Conference 2003, Warsaw, Poland, in: LNCS, vol. 2621, Springer, 2003, pp. 246–260.
[79] K.R.M. Leino, R. Stata, Checking object invariants, Tech. Rep. #1997-007, Digital Systems Research Center, Palo Alto, CA, Palo Alto, USA, 1997.
[80] D. Aspinall, K. MacKenzie, Mobile resource policies, in: M.H.G. Barthe, B. Gregoire, J.-L. Lanet (Eds.), Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: Proceedings of the Second International Workshop, CASSIS 2005, in: LNCS, vol. 3956, Springer, 2006, pp. 16–36.
[81] S. Sagiv (Ed.), Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, 4–8 April, 2005, Proceedings, in: LNCS, vol. 3444, Springer, 2005.
[82] C. Rattray, S. Maharaj, C. Shankland (Eds.), Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Proceedings, in: LNCS, vol. 3116, Springer, 2004.